A Model Driven Approach to Web Page Evaluation

Thomas Calder

June 2016

Master Thesis
The Department of Computer Engineering
Bergen University College
&
The Department of Informatic
University of Bergen

Supervisor: Professor Yngve Lamo

Abstract

We are in the midst of a technological shift from physical human-based services, to digital web-based services. As more commercial businesses and governmental agencies offer their services over the internet, a need for mandatory standards and guidelines has arisen in order to assure that the quality of the digital services are acceptable and accessible by everyone. However, as technology continues to progress, these guidelines change. Verifying that a digital service is up to date on the latest guidelines imposed by the government has become a daunting task because of several critical factors: Have the guidelines been understood correctly? Is everything that should be checked, being checked, and is it checked correctly?

In this thesis we will explore the possibilities of using *Model Driven Engineering* to first of all, formalize the mandatory requirements a web page must satisfy, so that it is possible to obtain a clear and concise definition. Second of all, using these formalized requirements, create a *Model Driven Web Evaluation Tool* that can evaluate existing web pages to determine if they successfully meet all the necessary requirements.

The results of this thesis will show that not only was it possible to create an automated web evaluation tool using Model Driven Engineering, but it has in fact overcome several acknowledged challenges and concerns related to modern day web evaluation tools.

Acknowledgment

I would like to give my sincerest thanks and appreciation to everyone who has contributed or helped me over the last two years.

First and foremost I would like give a special thanks to my supervisor Yngve Lamo for all the help and guidance he has given me. It has been at times extremely challenging working with this thesis, but my regular meetings with Yngve has kept me focused and motivated throughout the last year. I would also like to thank Fazle Rabbi for all the help he has provided me related to the WebDPF Editor.

Thank you Thormod Hellesøy and ACOS for giving me the opportunity to obtain my master's degree. Thormod has shown me a great deal patience and understanding over the last two years. I would also like thank my co-workers at ACOS for dedicating their time to helping me test my solution. A special thanks to Kent Inge Fagerland Simonsen for helping me find my topic for the thesis.

Finally I would like to thank my wife Elizabeth, words cannot express my gratitude for all the love, support and encouragement she has given me. I would also like to thank my sons Albert and Henry for their patience.

Bergen, 1st June 2016

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The internet has in recent years become one of the most important means of communication, replacing in certain situations, traditional forms of contact such as the telephone. Furthermore, web-based services are becoming a common substitute for many conventional over-the-counter services people were familiar with only a decade ago. Information provided by an organization can be easily accessed around the clock using web browsers such as Internet Explorer and Mozilla Firefox, making it unnecessary for individuals to contact that organization over the phone or in person during certain (and often inconvenient) hours. All sorts of services such as applying for loans from the bank, to booking plane tickets can also be accessed entirely through the internet almost instantly. At this rate the next generation might never have to know the frustration of having to stand in line waiting for their turn to be served.

This technological shift from physical human-based services to digital web-based services is not limited to the commercial sector however. The public sector has adopted many aspects of this trend, offering a great deal of services that can be accessed using the internet browser, such as applying for kindergarten or reporting municipal faults in your neighbourhood. A survey conducted by *The Agency for Public Management and eGovernment (DIFI)* [23] in 2014 showed that 31% of all governmental agencies offer some type of digital service [24].

By transferring many of these mundane services to the internet, the service and the process surrounding the service has generally become more efficient for *most* users, and there is the drawback with the digitalization of governmental and commercial services. Even though the common user is able enjoy the benefits of accessing their tax returns from the comfort of their own home, there is a minority group of people that are not able to use such services. This group consists of individuals who are visually impaired and therefore might have problems reading certain web pages. In more sever cases the individual might require an electronic aid such as a screen reader to interpret the contents of a web page. This group also contains individuals who are inexperienced with computers and the internet, such as the elderly. Common causes that might prevent this group from accessing these services are for instance lack of contrast between the textural content and its background, or replacing words such as *"menu"* with icons or symbols making certain interactions unintuitive to the user. Moreover, the web page needs

to be structured in a specific way in order for an electronic screen reader to read its contents.

In order to ensure that all Norwegian citizens may use digital services, the Norwegian government tasked DIFI with defining a set of criteria that will ensure that web pages are accessible to all users despite any disabilities or level of experience. DIFI is a governmental agency whose goal is to provide recommendations of standards for all Norwegian governmental websites and digital services. These standards primarily focus on making public information and services more accessible for residents and other public agencies [23].

1.1 Governmental Standards for Digital Services

There is no lack of governmental standards that web pages and other digital services are encouraged to satisfy or in some cases are obligated to fulfil. This section will provide a short overview of the criteria sets that a municipality or governmental agency must be consider when providing a digital service.

It is important that governmental digital services, especially digital forms, achieve a particular uniformed layout and interaction in order to ensure that these services are as intuitive as possible. This is covered by the *The ELMER Standards* [14], which is administered by *The Brønnøysund Register Centre* [13].

As emphasized in the introduction, it is also important to ensure that web pages and digital services can be accessed by all individuals regardless of any disability or level of experience they may have with computers and technology. DIFI provides a mandatory criteria set referred to as Universal Design criteria [25] that addresses these issues and provides a set of requirements all Norwegian web pages must satisfy.

Finally, it is important that digital services or web pages provided by a municipality or governmental agency maintains a certain level of quality. This is addressed in the optional criteria set *Kvalitet på nett* [26].

1.1.1 Universal Design Standard

The Universal Design standard aims at ensuring the main solution of the application accommodates as many people as possible regardless of disability. These mandatory requirements are applicable to all private Norwegian businesses and organizations as well as Norwegian governmental agencies and municipalities. The general idea is that designing the application to be as user friendly as possible, it will not be necessary to create any additional solutions specific for certain user groups such as the visually impaired, thus easier to maintain [25].

In order for a web page to successfully satisfy the Universal Design standard imposed by DIFI, the web page must fulfil the 35 A and AA criteria of the total 61 success criteria specified in

The Web Content Accessibility Guidelines 2.0 (WCAG 2.0) [56].

A survey conducted by DIFI in 2014 shows that the average Norwegian organization achieves only 54% of the total possible score when evaluated based on the Universal Design Standard and WCAG 2.0 [27].

The Web Content Accessibility Guidelines 2.0

WCAG 2.0 [56] was developed by the *Web Accessibility Initiative (WAI)* [55], as a means to provide a common set of standards that aids in making the contents of a web page more accessible to users with disabilities. These disabilities can range from visual and auditory, to learning and other neurological disabilities. WCAG 2.0 also attempts to make the web page content more accessible to older individuals with limited experience with web browsers and similar technologies.

At it's core, WCAG 2.0 consists of four principles it's guidelines build upon: *perceivable*, *operable*, *understandable* and *robust*. Moreover, each guideline contains a set of success criteria that describe how the web page can follow that guideline and in doing so, make its content more accessible to users regardless of any disability they may have.

The success criteria included in WCAG 2.0 are focused on important issues such as access for people with disabilities, and are defined in such a way that they are testable in order to confirm if the web page has fulfilled the criterion. These tests are conducted by a combination of machine and human evaluation.

WCAG 2.0 categorises its success criteria into 3 levels of conformance: *A (lowest)*, *AA* and *AAA (highest)*. There are a number of factors that determine the level of the success criteria: such as how *essential* it is, if it can be satisfied by *all web sites and types of content* and if it can be *reasonably be achieved by the content creators*.

It is the success criteria that determine if a web page is accessible to users with disabilities. In light of the WCAG 2.0 standard, the mandatory Universal Design standard described by DIFI can be defined as the following: *A web page conforms to Universal Design if and only if it satisfies all Success Criteria at level A or AA*

1.1.2 Other governmental standards

In addition to the mandatory Universal Design standard defined by DIFI, there exists other sets of criteria that aim at making a municipality's web pages and digital services more efficient and user friendly. These requirements are not mandatory, but are encouraged to fulfil.

The ELMER Standards

ELMER 2 (which is a Norwegian acronym for Simpler and More Efficient Reporting) is a set of guidelines which ensures that digital forms are designed in a uniform and user-friendly way. These standards are generally focused on the layout and process of filling out the digital form and less on the actual implementation of the html elements that the form consists of [14].

An example from the ELMER 2 standards is requirement 2.3.1 that states that every page in a form must contain its own page title which describes the general theme of the current page, such as "Applicant" for a page where the applicant provides details about him- or herself. It does not specify how this should be implemented, the developer is free to use which ever html element he or she likes [14].

The requirements specified in the ELMER 2 standards are applicable to digital forms provided by a Norwegian governmental agency or a municipality.

Kvalitet på nett

The *Kvalitet på nett* standard focuses more on the users interaction with the web page or digital service, rather than how it has been implemented. The standard consists of 33 criteria based on other acknowledged standards and guidelines, as well as political guidelines [26].

Typically the sort of requirements this standard contains, is for instance how easy it is for the user to interact with the web page with regards to the layout, security and quality, or how well certain digital services works.

1.2 Motivation

As the internet becomes an increasingly more important tool for communication, it is necessary to ensure that not only is the quality of digital services provided by governmental agencies and municipalities adequate, but also equally available to all its users. So as web based technology continues to develop rapidly, the need arises for automated tools that can ensure that web pages and digital services comply with the latest guidelines or mandatory criteria imposed by the authorities. This is something that can benefit not only users with disabilities, but also the programmers who must develop web-based applications that fulfil these requirements.

This paper will describe an attempt to use methods and techniques from *Model Driven Engineering*[10] to create an automated web evaluation tool that can be used to determine if a web page satisfies the necessary requirements imposed upon it, and thus resolve if the web page is valid or not.

This will be attempted by designing a modelling language that can describe the mandatory

requirements a web page must fulfil. The modelling language will be formalized as a meta-model that will be used as the core of a developed web evaluation tool that will attempt to confirm if a web page satisfies all the mandatory requirements included in the evaluation.

The success of this thesis will be evaluated based on two factors; firstly by how well the solution handles several problems and challenges described in chapter 2 and how it fares when compared to other existing web evaluation tools.

Finally, the results from this thesis will include two things: first of all, it will provide an alternative solution to the existing problems and challenges related to automated web evaluation. Second of all, although there already exists automated web evaluation tools, there are none to best of the author's knowledge that attempt to validate a web page using techniques from the Model Driven Engineering. Thus this thesis introduces a new possible application for the Model Driven Engineering methodology.

1.3 Structure of thesis

Chapter 2 will describe what sort of problems and challenges involving automated web page evaluation that will be addressed, as well as present a set of requirements that the solution in this thesis will use when validating a web page.

Chapter 3 provides an explanation of the background theories that are essential for this thesis. The concepts of Model Driven Engineering, Domain-specific modelling languages, the Meta-Object Facility and the Diagram Predicate Framework are described, in order to prepare to reader for the next chapters.

In **Chapter 4** a formal definition of the selected requirements is described with the help of a Domain Specific Modelling Language. This definition is represented as a model that will serve as the foundation for the automated web evaluation tool, that will attempt to confirm if a web page fulfils the selected requirements.

Chapter 5 describes two possible solutions that involves using the model from Chapter 4 to evaluate a web page. These two alternatives are referred to as the *Top-down Model Validation approach*, and the *Bottom-up Model Validation approach*. Chapter 5 will conclude with a discussion on which solution is best suited for evaluating existing web pages.

In **Chapter 6** the preferred solution from Chapter 5 will be evaluated based on how well it overcomes the challenges described in chapter 2. This will be done by conducting a small usability experiment, as well as comparing the solution with other existing web evaluation tools.

Chapter 7 will end this thesis with a conclusion and a short discussion of limitations of the solution and potential future work.

Chapter 2

Problem Description

This chapter will describe several concerns this project will attempt to overcome when validating a web page automatically using a web evaluation tool. First there will be a definition of what a web evaluation tool is, followed by a description of each challenge this paper will address. The chapter will conclude with a explanation of which requirements will be included in the solution.

2.1 Web Evaluation Tools

A web evaluation tool can be defined as an application that inspects if a web page satisfies a set of requirements. Any element contained within the web page that violates one or more of the requirements is flagged as invalid, and reported to the user. Some of the more advanced evaluation tools suggests possible corrections for the invalid element.

There exists a number of web evaluation tools already that are able to determine if a web page is valid based on a set of requirements. However, many of these tools are limited to inputting a web page, and outputting the results, revealing little information concerning the actual evaluation process [29]. The problem with this is that a user has to rely upon the fact that the tool is evaluating the web page correctly. This leads to several questions: What is actually being evaluated and is it being evaluated correctly? How can the user be sure that the results of the evaluation are correct? These questions originate from a common problem: How can the user efficiently measure the effectiveness and legitimacy of the tool?

2.1.1 Challenges

Evaluating the effectiveness and legitimacy of a web accessibility tool could include many different aspects ranging from the time it takes to evaluate a web page, to the accuracy of the results presented by the tool or how it handles fault identification and diagnosis. This thesis will at-

tempt to overcome several challenges related to the problem described above.

Furthermore, developing an adequate web evaluation tool is not simple when considering the fact that the expert within a certain criteria set, hereby referred to as the domain expert, might not have the necessary skills to develop the tool themselves, and must therefore rely upon the help of a developer who may not be familiar with the domain. The common root of many of the challenges presented in this section involve providing the means for the domain expert to confirm that the developer has interpreted the requirements correctly, and thereby created a web evaluation tool that evaluates the web page correctly.

The transparency problem and *the barrier problem* were first mentioned in [29], and addresses the concerns linked to the lack of insight in how an automated web evaluation tool evaluated a web page.

The correctness problem, *the completeness problem* and *the specificity problem* were described in [9] as three important aspects when evaluating an automated web evaluation tool or for comparing web evaluation tools.

The remaining challenges: *the Generic Evaluation Problem*, *the Ambiguity Problem* and *the Modification Problem* are introduced in this thesis based on potential challenges this project may come across. This section will describe these challenges.

The Transparency Problem

The first challenge involves providing the means for an expert to determine if the tool is "*checking everything that it should be checking correctly*" [29]. In other words, the lack of insight in how the requirements are interpreted and used in the evaluation process makes it difficult for the domain expert to confirm if the web evaluation tool is generating accurate results. Furthermore, it is not adequate to only provide documentation on what is included in the evaluation process or how the process is executed, since it only reflects what is included when written and may easily become outdated as soon as the process is changed. What the domain expert needs is the ability to observe the actual evaluation process.

This is a particularly tricky problem to overcome since it first of all requires access to the procedure that evaluates the web page, and commercial web evaluation tools are probably not going to want to share that. Second of all, the evaluation process most likely consists of functions that the domain expert might have difficulties understanding (i.e the domain expert is not a developer), the domain expert will need to rely on someone else to explain the procedure. This can lead to other problems related to misunderstandings.

An adequate solution to the problem would be executing the evaluation process in a way that can be interpreted by domain experts without programming knowledge.

The Barrier Problem

The second challenge is managing potential barriers the tool might come across during the evaluation process. A barrier is defined as anywhere in the evaluation process where a human must make a decision, an example of such as barrier is for instance determining if a header describes the contents of its corresponding section well enough [29].

The Completeness Problem

The third challenge is determining how complete the web evaluation tool is. Completeness can be defined as a measure of how many violations are detected by the web evaluation tool when inspecting a web page. An incomplete tool might not be able to detect elements that do not satisfy the necessary requirements, leading to the problem of *false negatives*, which occurs when violations are not identified and flagged. In other words, one could say that completeness is related to the tools ability to reduce false negatives [9].

The consequence of not managing false negatives is that the tool wrongfully reports that a web page containing errors, successfully conforms to the requirements specified in WCAG 2.0. Unfortunately it is difficult to determine if the evaluation of a web page contains any false negatives without implementing additional tools that allows an individual to confirm that the web page is evaluated correctly [9]. Worst case scenario this could require a developer to inspect the evaluation process, element-by-element using a debugging tool.

The completeness problem addresses how many and how thorough the requirements interpreted by the web page evaluation tool are. The more requirements that are implemented into the evaluation tool, will lead to less false negatives contained in the results generated by the evaluation process. More importantly, by informing the user which requirements are included in the evaluation process, the easier it will be for the user to discover what is not being evaluate and therefore easier to detect potential false negatives.

The Correctness Problem

The fourth challenge is checking how correct the results of the web evaluation tool are. Correctness can be used to measure how many of the identified violations are actually problems, referred to as *false positives*. False positives pollute the result of the evaluation process with errors that do not necessarily break any of the requirements. The level of correctness can be determined by how well the tool is able to reduce false positives. [9]

Some evaluation tools deliberately implement false positives as warnings in situations where there may be a potential violation, the problem with this is that the evaluation tool gives the impression that there are potential errors in a web page that adequately satisfies the necessary requirements. Additionally it makes it difficult for the user to determine which warnings should

be investigated and which can be safely ignored [9].

The correctness problem addresses how well the web page evaluation tool minimises false positives during the evaluation process, as well as provide the user the means to verify if the results contain any false positives.

The Specificity Problem

The fifth challenges involves how specific the results from the web evaluation tool are. Specificity measures the level of detail the tool describes when reporting a violation detected in the web page. The more detailed the violation is described, the more useful it is for the developer. It isn't sufficient to state that an input element is missing a descriptive text, a warning explaining that the input element requires a title attribute or referred to by a label element would aid the developer in correcting the error. In other words, a warning should not only identify a violation, but also detect the root cause of the violation and guide the developer towards a correct solution [9].

The specificity problem focuses on how detailed and useful the feedback from the web evaluation tool is.

The Generic Evaluation Problem

The sixth challenge is concerned with the ability to evaluate different representations or components of a web page. Generally a rendered web page is defined using HTML, but any styling the web page contains requires styling rules implemented with *Cascade Styling Sheets* (CSS), and user interaction is resolved using Javascript or any number of server side languages such as C#, Java or PHP. In order to validate all necessary aspects of a web page, it is reasonable to assume that the web evaluation tool should not be limited to one technology or platform. Furthermore, in order to avoid creating separate evaluation processes for each component, the evaluation process should attempt to be as generic as possible.

The Generic Evaluation Problem addresses the reusability of the different components included in the evaluation process.

The Ambiguity Problem

The seventh challenge involves the possibility of misunderstandings that may occur when interpreting the requirements. The different standards described in the introduction are specified using a natural language, even though some of the criteria are described in detail with examples, others are not so specific leading to the danger of ambiguity. Either way, the fact that the

requirements are defined in a natural language means there will always be a danger of misinterpretation.

Furthermore, the requirements a web page needs to satisfy could be defined by a project manager with expertise within a relevant domain, these requirements might then handled by a developer who needs to ensure that the solution he or she developed is correct. Problems related to ambiguity might occur if the project manager is unable to describe these requirements in a format the developer can understand. In other words, it is vital that the requirements are defined in such a way that they build up a common understanding.

The Ambiguity Problem focuses on the precision of the requirement's definition and the ability to communicate it's contents between individuals with different areas of expertise.

The Modification Problem

The eighth and final challenge involves handling alterations made to the requirements. The criteria might change frequently. DIFI re-evaluates their criteria annually, this involves adding, removing and editing existing criteria. With the previous challenges in mind, it is important to ensure that modifications are correctly implemented, changes to the requirements should be limited to an environment that experts within the domain can perceive, and thus confirm that the evaluation tool is checking web pages correctly.

It should also be noted that many of the existing web evaluation tools are based on international standards such as WCAG 2.0, that may not coincide entirely with what the user wants. An example of this is potential recommendations from *Norges Blindeforbund* [6], a Norwegian organization that represents the interests of the visually impaired, that may vary from the specifications included in WCAG 2.0. The Web Evaluation tool should be flexible enough to be able to permit modifications to the requirements, that can be applied by the domain expert, who does not necessarily have the knowledge or the means to make changes to the tools source code.

The Modification problem addresses two concerns: first of all any modifications made to the requirements need to be done in a way that allows an expert to confirm its validity. Second of all, the web evaluation tool should provide the means to allow the domain expert to make necessary modifications ideally without involving any changes to the source code.

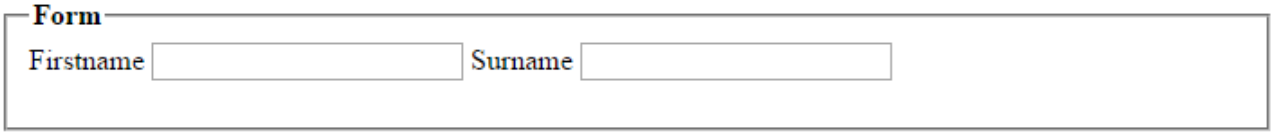
2.2 Selected requirements

The solution presented in this thesis will in theory be able to validate a web page based on any of the requirements specified in the introduction, but in order to avoid mixing in a lot of requirements belonging to different sets, the solution will only include a selected list of requirements from the WCAG 2.0 specification. See table 2.1 for an overview of which requirements that will be included.

Requirement	Type	Mandatory	Criteria level
1.1.1 Non-text Content	Images and graphics	Yes	A
1.4.5 Images of Text	Images and graphics	Yes	AA
1.4.9 Images of Text (No Exception)	Images and graphics	No	AAA
1.3.1 Info and Relationships	Text and structure	Yes	A
2.4.6 Headings and Labels	Text and structure	Yes	AA
1.4.8 Visual Presentation	Text and structure	No	AAA
2.4.10 Section Headings	Text and structure	No	AAA
3.3.2 Labels or Instruction	Text and structure	Yes	A
1.4.3 Contrast (Minimum)	Contrast and Styling	Yes	AA
1.4.6 Contrast (Enhanced)	Contrast and Styling	No	AAA

Table 2.1: Table of selected requirements [56]

By focusing on the requirements belonging to the WCAG 2.0 specification, the solution presented in this thesis can be categorized as a *Web Accessibility Evaluation tool*, and thus when evaluating the solution, it can be compared to other existing Web Accessibility Evaluation tools.




```
<fieldset>
    <legend><strong>Form</strong></legend>
    <form>
        <label for="inputFirstname">Firstname</label>
        <input type="text" id="inputFirstname" />
        <label for="inputSurname">Surname</label>
        <input type="text" id="inputSurname" />
    </form>
</fieldset>
```

Figure 2.1: Input elements are labelled correctly using the *for*-attribute

An example of one of the requirements specified in WCAG 2.0 is *2.4.6 Headings and Labels* that states among other things, an input element in a digital form must have some type of description attached to it. This can be solved using either a label element which is correctly attached to the input element using the labels for-attribute, or using the inputs title-attribute. Figure 2.1 shows an example of a correct form and figure 2.2 shows an incorrect form. [18]

Another example is the requirement *1.4.3 Contrast (Minimum)* and *1.4.6 Contrast (Enhanced)*, which states that the contrast between the web page's text and background is satisfactory. The minimum contrast ratio is set at 4.5:1, but 7:1 is the recommended ratio. Figure 2.3 shows an example of the contrast ratio. The remaining requirements in table 2.1 will be explained in chapter 4.


```
<fieldset>
    <legend><strong>Form</strong></legend>
    <form>
        <input type="text" id="inputFirstname" placeholder="Firstname"/>
        <input type="text" id="inputSurname" placeholder="Surname"/>
    </form>
</fieldset>
```

Figure 2.2: Input elements are incorrectly implemented since they are using placeholders instead of labels or the *title*-attribute

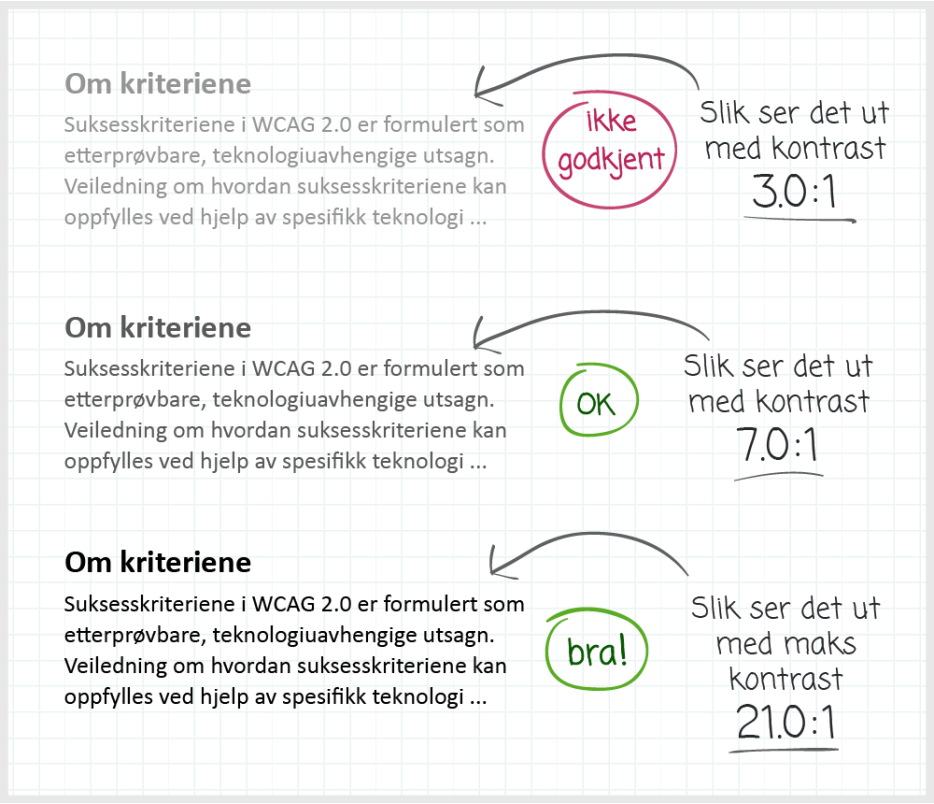


Figure 2.3: Examples of the different levels of constrast. 3.0 1 is rated as "Not Approved", 7.0 1 is rated as "Okay", 21.0 1 is rated as "Good"

Chapter 3

Theoretical Background

In this chapter the background theory for this thesis will be presented. It will start of by explaining what Model Driven Software Engineering is, followed by an explanation of what Domain-Specific Modelling Languages are. This chapter will conclude with a description of two meta-modelling frameworks: The Meta-Object Facility and The Diagram Predicate Framework.

3.1 Model Driven Software Engineering

Model-driven engineering (MDE) is a methodology which focuses primarily on two things: models and transformation of models. The model becomes the most central artefact during the development process in that everything is treated as a model. *Model Driven Software Engineering (MDSE)* applies the methods and techniques defined in MDE to software engineering activities. This provides mechanisms such as model validation, model transformation and model generation. All which contribute to an overall higher quality of the developed system [10].

Creating model representations of software systems is not a new concept. Software development processes such as *the rational unified process* [34] encourage developers to produce large amounts of documentation including model representations of the system to be developed. The model provides a better understanding of complex systems for the developers and other members of the team, but serves as little more than a piece of documentation that needs to be maintained and updated during the development process. MDSE introduces tools that make the model executable.

One of the most influential driving forces behind MDSE is the *Object Management Group (OMG)*. OMG is a consortium who's main purpose is to set and provide standards in object-orientated programming. Their primary goals are the reusability, portability and interoperability of object-based software components [45]. OMG has provided several standards such as *The Common Object Request Broker Architecture (CORBA)* and *the Model Driven Architecture (MDA)*. Note that OMG only provides standards and specifications and not any implementations.

This section will explain the core principles and concepts of MDSE as well as some central specifications defined by OMG.

3.1.1 Core Principles and Concepts

At its core, MDSE consists of models and the manipulation of models (commonly referred to as model transformations). These two concepts are used to describe software from a MDSE point of view with the following algorithm [11]:

Models + Transformations = Software

MDSE describes models as a simplified version of a software system or components within the software system. The defined model may be used for descriptive purposes where it represents an existing system, or for prescriptive purposes where a system is generated from the model [10]. The model is generally considered to be a more abstract view of the system where only selected parts of the system is reflected in the model, and the rest is removed by abstraction. Section 3.2 will explain how models can be used to represent or describe domain-specific aspects of a system [31].

Models are not limited to describing systems, but can in fact describe other models. These models are called meta-models and are basically models that contain the definition to construct other models. Meta-models were originally used to define schemas for semantic data that needed to be exchanged. A meta model could for instance be a *XML Schema Definition (XSD)* describing a *XML* that is shared between multiple systems [20]. In the world of modelling a meta-model is considered an abstraction of model where the model's properties are highlighted. It can be described as a model of models [10]. We can generally use meta-models to define modelling languages since they provide a way of describing a whole class of models. A valid model always conforms to it's meta-model. Figure 3.1 shows an example of the relationship between models and meta-models [10].

There are two types of models defined: *static models* and *dynamic models*. Static models focus on the static aspects of the system such as the structure, and dynamic models focus on the dynamic behaviour of the system [10]. A static model of a web page would represent for instance the structure of a web page including elements like labels, inputs and so on, a dynamic model would however focus on how the web page reacts to user interaction. This might include the behaviour of the web pages javascript or server side responses to requests from the user.

MDSE uses transformations to produce other models and artefacts (i.e. a model of lower abstraction or textual output such as executable source code) from a model, this is achieved by using transformation patterns or mapping rules. Transformations allow multiple representations of a model to created across different levels of abstraction and from different views [31]. Furthermore, model transformation can aid a developer by automating certain tasks, which con-

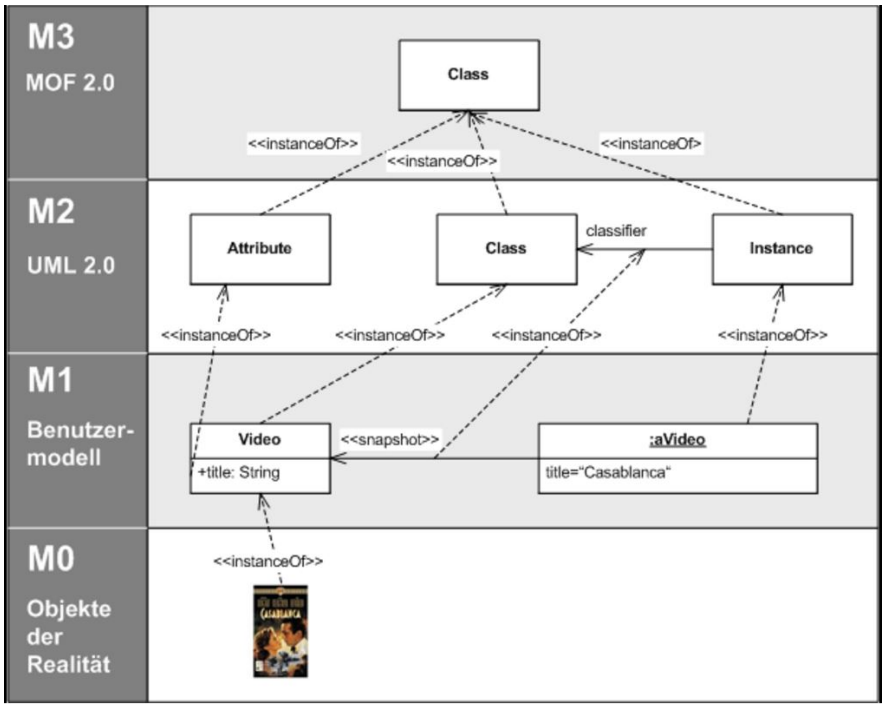


Figure 3.1: Example of the relationship between models, meta-models and meta-meta-models

tributes to a more efficient and productive development process [10]. There is a more detailed explanation in section 3.1.4.

One of the main benefits with using MDSE is that it provides abstraction from specific platforms and technologies by using notations that define the models and the transformations. This is commonly referred to as the *modelling language* and improves the portability of the software by making it possible for the developer to create a single model that may be used in multiple instances [10].

Finally, there are two core relations between models defined by MDSE: *Representation* and *Conformance*. These relations correspond to inheritance and instantiation that are found in object-orientated programming languages such as C# and Java [10].

3.1.2 Model-Driven Architecture

In order to handle the complexity and extract data from large systems, OMG specified the *Model-Driven Architecture (MDA)* framework. The MDA framework should however not be considered as a separate OMG specification, but rather an approach to system development which is enabled by existing OMG specifications such as the Meta-Object Facility, that will be explained later in section 3.3. MDA builds upon the several principles that underlie the OMG's view of MDE [10]:

- Models must be expressed in a well-defined notation. This is to achieve a common understanding of the model and in doing so improve communication and understanding of the system
- The Systems specifications must be organized around a set of models and associated transformations implementing mappings and relations between the models.
- Models must be built in compliance with a set of meta-models, thus facilitating meaningful integration and transformation among models and automation through tools.

One of the main purposes of the MDA framework is to obtain information from a model, the MDA specification document suggests several ways to interpret information from models [31]:

- *Models as a communication medium* - The model aids in reaching a common understanding within the team. MDA provides the means of developing a set of well-defined terms and notations specific to the domain the team is operating within, this is commonly referred to as a *Domain-Specific Modelling Language (DSML)* which is a modelling language tailored for a specific domain. This will discussed in section 3.2.
- *Derivation via automated transformation* - Model transformation allows platform independent "source models" to be transformed into a platform specific "target artefact". This could be for instance transforming a model to a HTML document in order to present some information within the model, or using the model to generate a data structure intended for a Java or C# application or even a database. Code generation from models is considered one of MDAs primary uses.
- *Model Analytics* - By representing data as a model, it is possible to use MDSE techniques and methods on that model such as model validation and statistics. This contributes to improving the overall quality of the system.
- *Model Execution* - MDA is not limited to transforming source models to code, Model Execution provides the option of directly using a model with minimum technical information being exposed. Models can be used directly in high-level applications, for instance serving as a repository for the the applications configurations.
- *Structuring Unstructured Information* - Information extracted from an unstructured document can be stored in a model, this provides a formal structure to the information and may provide a better understanding of the documents content and message.

The Modelling Levels

The MDA Framework introduces three levels of abstraction a model can be represented in. The abstraction levels are organized into a hierarchy with the most abstract level at the top, and each descending level being more detailed. The abstraction levels doesn't need to be limited to three levels, but normally every level in the hierarchy will have the characteristics of one of the

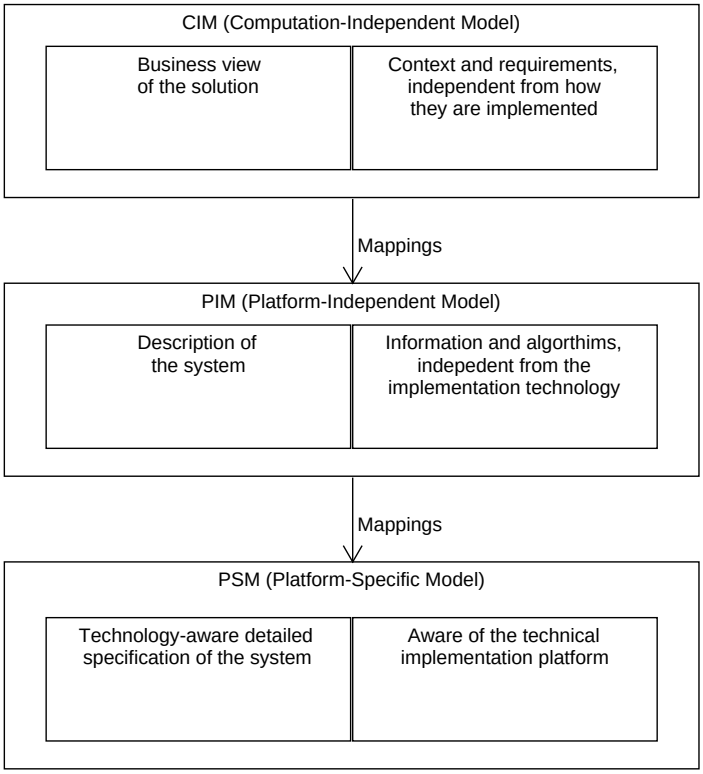


Figure 3.2: The three abstraction levels defined by MDA

levels described by MDA. Between each level exists a mapping from the higher abstraction to the lower abstraction. These mappings can be viewed as model transformations between two adjacent models in the hierarchy. The level of abstraction is determined based on what is being modelled, meaning that there may occur some deviation from the level descriptions below [10]. Figure 3.2 shows the three levels of abstraction and how they relate to each other [10].

The Computation-Independent Model (CIM) is the most abstract model and positioned at the top of the hierarchy. It normally represents the context, requirements and purpose of the system, but doesn't include any details on how the system should be implemented. This is to keep the model independent of any platform or technology. Models at this level are commonly referred to as business models or domain models since they consist of terms and notations belonging to a specific domain. Note that the CIM isn't necessarily limited to software-based implementations, some parts may be intended to stay at this level of abstraction such as documentation composed in a natural language [10].

The Platform-Independent Model (PIM) is a model that describes the behaviour and structure of the system, but remains platform independent. At this level the model consists of information and algorithms, but nothing that will tie it to a specific technology. These models are often referred to as Logical System models since they generally focus on the way the components of system interact with each other, without including platform specific data [31]. This way the model may be mapped to multiple technologies such as Java, C# and JavaScript. It is

only the software-based part of the CIM that is mapped to this level, documentations and other artefacts that are not software-based won't be represented at this level of abstraction [10].

The Platform-Specific Model (PSM) is at the bottom of the hierarchy and as the name suggests the model consists of information about the behaviour and structure of the system specified in such a way that it is specific to a single platform [10]. These models are referred to as implementation models since they model how the system is to be implemented on a specific platform in order to carry out its function. [31]

The purpose of the three level architecture is to define or improve systems. The architectural process encompasses: understanding the scope of the systems of interest, understanding requirements, and arriving at a design to satisfy those requirements. By using MDA's modelling levels it is possible to understand and specify a system while ignoring details such as how it should be implemented in a certain platforms or technologies. The further up the hierarchy the model is positioned, the more systems it can represent or is applicable to. This means that the CIM and PIM can be mapped to multiple models, and in doing so improves the reusabiliy of the model.[31]

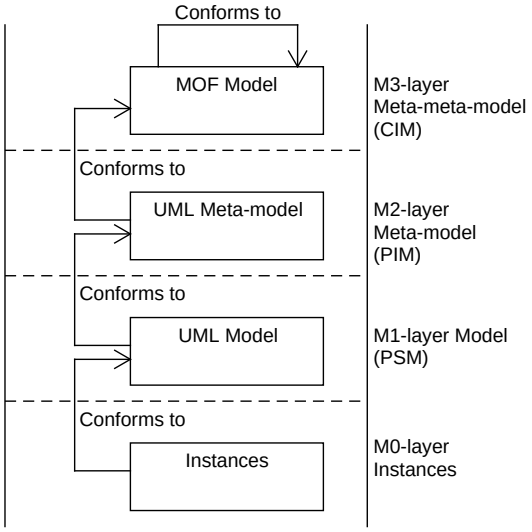


Figure 3.3: The four layers of a MOF-based meta-modelling hierarchy

The Modelling Levels defined in this section can be found in meta-modelling hierarchies. Figure 3.3 shows how a typical MOF-based framework for meta-modelling consisting of four layers is structured. These layers are described as followed [37]:

- *The M0-layer* consists of the information that needs to be described. This is commonly referred to as "data".
- *The M1-layer* contains the models that describes the information.

- *The M2-layer* is comprised of meta-models that provide descriptions and definitions for the models structure and semantics.
- *The M3-layer* consists of the description and the definition for the meta-model. It can be considered as a modelling language for modelling languages. It is also reflective, which means it is able to describe itself and thus serve as a natural end to the meta-modelling hierarchy.

3.1.3 Software Reengineering

A common problem that almost all software systems are exposed to at some point, is when a legacy application is not running optimally in a modern system, a problem referred to as *Software Decay*. A common way to prevent software decay is to restructure the legacy system using modern technology, this process is called software reengineering. However, most software reengineering projects fail due to lack of a standardization and automation in the reengineering process [40].

The *Architecture-Driven Modernization (ADM)* is a concept developed by the *Architecture-Driven Modernization Task Force (ADMTF)* to handle problems related to software decay and solve the challenges linked to software reengineering. ADM does this by introducing a formal process for reengineering software based on the MDA standard [40].

The reengineering process consists of three stages [40]. Figure 3.4 shows the reengineering process defined by ADM [42]:

- *Reverse Engineering* - The first stage involves analysing the legacy system (or source environment) in order to identify components and the relationships between the components. One or more higher level models are then generated using the components identified in the legacy system. These models are abstract representations of the legacy system. In the context of MDA, the process is basically mapping the PSM to the PIM and then mapping the PIM to the CIM.
- *Restructuring* - The second stage executes a number of model transformations upon the abstract models generated in the previous stage. The restructuring process ensures that the external behaviour of the legacy system is retained.
- *Forward Engineering* - The final stage of the process follows the MDA process. It uses the result (i.e. the model generated by the model transformation) from the restructuring stage to create a PSM in the target environment.

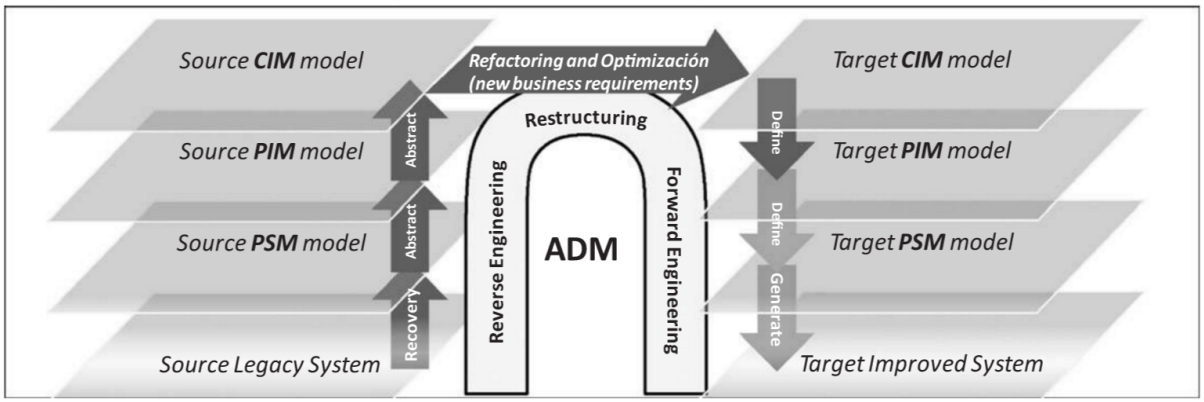


Figure 3.4: The Horseshoe model

The Knowledge Discovery Model

One of the challenges with the reengineering process is automating the creation of the PIM model during the Reverse Engineering stage. As explained previously, the PIM is an abstract representation of the PSM. In other words it can be considered to be the meta-model of the components in the legacy system. The question is then, how is it possible to automate the process of creating the meta-model? ADM introduces the *Knowledge Discovery Model* to serve as the initial meta-model (or PIM) during the reverse engineering stage of the reengineering process. [41]

The KDM's main goal is to provide a comprehensive view of the legacy system, as far down as the procedure level. It represents artefacts of the legacy system such as entities, relationships and attributes as well as external artefacts the legacy system interacts with [41]. Since the KDM is platform and language independent at its core, it has the same characteristics as the PIM model and therefore can serve as a model at this level of abstraction.

In the Reverse enginnering stage of the reengineering process, knowledge is extracted from the KDM by executing a series of KDM-to-KDM transformations. Figure 3.5 shows an example of the KDM's role in the reverse engineering stage.

The KDM standard provides a higher-level abstract view of the behaviour, structure and data components belonging to an artefact. These standards basically consist of meta-models describing a piece of information (i.e. part of a legacy system or web page) belonging to the artefact, and how these meta-models can be represented using OMG standards such as MOF and XMI. The KDM standard refer to these meta-models as facts [41]. The KDM meta-models provide the means to first of all, store information about the artefacts components in a common structure. Secondly the information can be analysed and evaluated using techniques from MDA. Thirdly, since the meta-model can be represented using XMI, it can be exchanged with other reverse engineering tools that utilize KDM. Finally these meta-models can be used to build parsing tools to extract platform-independent models from the artefact [41].

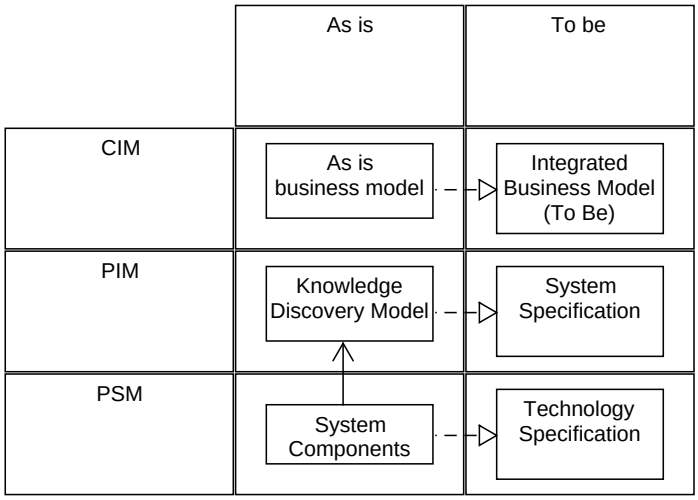


Figure 3.5: Example of a reverse engineering scenario [10]

The KDM is divided into four layers [41]: *The Infrastructure layer*, *The Program Elements layer*, *The Runtime Resources layer* and the *The Abstractions layer*. Furthermore each layer consists of several packages used to define meta-model elements that represent specific parts of the artefact. See figure 3.6 [35].

The Infrastructure layer is the lowest level of abstraction and consists of the Core, KDM and source packages. These packages define the common fundamental meta-model elements that are used in the other KDM packages found in the higher level layers [35].

The Program Elements layer contains the Code and Action packages. The code package focuses on elements found in the source code, and the relationships between the different elements. The action package is used by other KDM packages to describe the artefacts behaviour at a higher level of abstraction [35]. The Program Elements layer basically represents the logical view of the artefact [41].

The Runtime Resource Layer consists of the Platform, UI, Event and Data packages. These packages focus on the information that is not contained within the source code, but rather the information managed by the artefact, such as the database or user triggered events [41].

The Abstraction layer is the most abstract layer in the KDM and contains the Structure, Conceptual and Build packages. The main purpose of this layer it to provide a representation of the domain-specific knowledge and a business-overview of the artefact [41].

Each of these layers would normally contain vast amounts of information during the reverse engineering process, making it difficult for a domain expert or developer to interpret, therefore each layer has the ability to generate models of itself at different levels of abstraction to improve understanding of their contents [35].

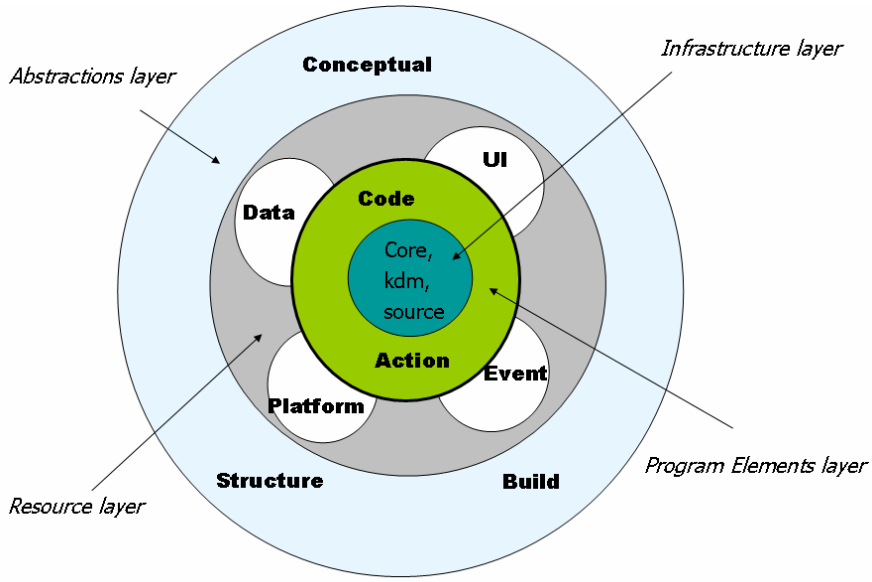


Figure 3.6: The Knowledge Discovery Meta-model is organized into four layers: *The Infrastructure layer*, *the Program Elements Layer*, *The Runtime Resources layer* and the *The Abstractions layer* .

3.1.4 Model Transformation

As previously explained, all operations executed upon models are implemented as transformations. Model transformations are described as definitions at the meta-model level which provides mappings between different models. The transformations are then applied to the models that conform to the meta-models [10]. Generally the conceptual content of the model stays the same, but is presented in another form [31].

There are two general types of model transformations: *Model-to-model (M2M)* and *model-to-text (M2T)*. A M2M transformation is a program that takes in one or more models and produces one or more models. The transformation can either be executed as an exogenous (out-place) transformation where a new model is generated, or endogenous (in-place) transformation where the original model is rewritten [10]. M2M transformations can be used to merge, refactor, refine and convert models. For instance MDA model transformations are used to transform CIMs to PIMs, and PIMs to PSMs, this is basically "forward engineering". ADM attempts to do this in the opposite direction by using M2M transformations to transform PSMs to PIMs.

M2T transformations are programs that take in one or more models and outputs something textual. These transformations are mainly used for code generation and therefore often used as the final stage in a chain of model transformations. The end goal of MDSE is to create executable components or applications from models, this is achieved using M2T since the generated textual output can be for instance runnable source code or a web page [10]. Figure 3.7 illustrates how a chain of model transformations might look. M2T can also be mapped the opposite way as a *Text-to-Model (T2M)* transformation, in these scenarios models are extracted from textual inputs like

documents and source code.

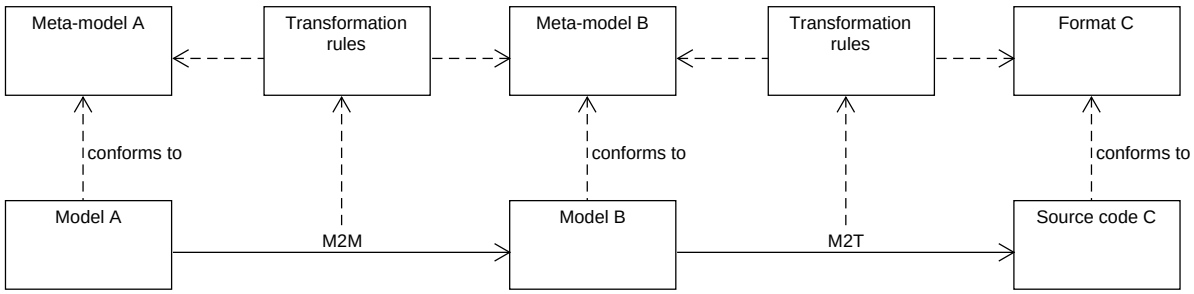


Figure 3.7: An example of a model transformation process

The term *mapping* has been used several times throughout this section. A mapping is a link between two different meta-models, it basically consists of rules and algorithms for how a valid model transformation between two meta-models is executed. A mapping between two meta-models may be applied to their instances as well. There are two types of mappings; intensional which is a mapping at the meta-model level, and extensional which is a mapping defined at the model level. Mapping specifications can be be defined as *weavings* which are simple correspondences between meta-model elements, or as *constraints* that describes the requirements for the model transformation. The main objective of model mapping is to automate the transformation between models. [10]

3.1.5 Model Validation

Meta-models are used to define the abstract syntax and constraints that define the well-formedness of a modelling language. A model that is a valid instance of it's meta-model is considered a valid model, this is commonly referred to as conformance. A model that doesn't conform to it's meta-model is not considered as a valid instance and therefore is invalid [10].

Before going any further, it should be noted that in the context of MDSE and *Model Quality*, model conformance is just scratching the surface of model validation and verification. Models that conform to their meta-model may still be invalid, this is addressed in *The satisfiability problem* where it is impossible to instantiate the model in a way that satisfies all of its constraints [10]. These sorts of problems are rather complex and outside the scope of the challenges presented in chapter 2. Therefore a valid model will be defined as a model that is correctly typed and satisfies all it its constraints.

3.2 Domain-specific Modelling Languages

In order to serve its purpose, the model needs to be defined in such a way that it relays its information correctly and in doing so, provide a better and common understanding of the information it contains to all interested parties. As mentioned in the previous section, a modelling language makes a model easier to interpret by providing a well defined and consistent structure and set of notations, terms, syntax and semantics [31]. Furthermore a modelling language allows developers to define concrete models in their system that consists of graphical representations and textual specifications. The modelling language may be described using meta-models that represent the central concepts of the language [10]. MDSE describes two types of modelling languages: *Domain-specific modelling languages* and *General-purpose modelling languages*.

General-purpose Modelling Languages (GPMLs) are languages that may be applied to any domain or context. GSMLs are normally graph based with limited support for constraints. Since the semantics are often general and imprecise, automating any code generation is considered difficult. The most well known example of a GPML is *The Unified Modelling Language (UML)*. [10]

Domain-specific Modelling Languages (DSMLs) are languages designed for a specific domain or context making it easier to communicate with people who operate within that domain. Normally DSMLs are specified by graph-based meta-models combined with text-based constraints. DSMLs differ from GPMLs by being less expressive, yet it describes the selected domain more clearly and concise [10]. A DSML can be for instance an XML document containing an applications configurations. This section will focus primarily on DSMLs.

Using a DSML provides several advantages. Firstly it improves development productivity by providing a clear definition of the system. Since the DSML has a limited expressiveness it describes its components more concise and leaves little room for ambiguity and faulty explanations. It also provides a level of abstraction that filters out irrelevant elements and components making it easier to specify what's going on and detect potential errors. Secondly since the language is clear and concise, it improves communication between domain experts and developers. A DSML makes it easier for a domain expert to spot mistakes in the system specified by that DSML and instruct a developer of the necessary corrections. Obviously this depends on the context, not all DSMLs are suitable for communicating. Thirdly by defining a system with a DSML, it is possible to represent that system as a model and run it in multiple environments by executing it directly or generating code from it [28].

One last advantage with DSMLs that is worth mentioning is the conformance check between a model and it's modelling language. A formal definition of a DSML can be represented using a meta-model, this means it can be determined if the model is a valid instance of the DSML, this is in contrast to an informal modelling language (such as the ones written on a white board during a discussion) that lacks any way to check for conformance and so is easily misinterpreted or forgotten [31].

By defining a DSML for the web page domain, it is possible to not only describe accurately

a valid web page, but also provide the means for domain experts to verify that the requirements the web page must fulfil are correctly interpreted and implemented by the developer. Furthermore by using separate DSMLs for the different components that the web page consists of, it is easy to avoid ambiguity problems by differentiating between common terms such as "attributes", this is not something we could easily achieve using a GPML.

DSMLs do however have some drawbacks. First of all there is the problem of *Language Cacophony* which is concerned with the problems of learning a new language. It is also not unusual for a system to use multiple DSMLs (as we saw above) to describe the different components it is comprised of. This might contribute to some confusion and it may become difficult to understand the overall process or operation of the system. The second drawback is the *Cost of Building* the DSML. Building a DSML on top of a component may be a small cost, but there is still a cost and it still has to be maintained. Lastly there is the problem of creating a *Ghetto Language*, which occurs when a DSML is built on an in-house language. When the language exists solely within the system, it becomes difficult to introduce new developers to the system or keep the system up to date with technological changes [28].

Despite it's drawbacks, defining a modelling language as a DSML is better suited, than using a GPML, since a DSML is tailored to satisfy the requirements of a specific domain [10]. The clear and precise language could provide a solution to the ambiguity problem described in 2.

DSMLs can be divided into two categories [28]:

- *External DSML*: is a language separate from the main language the application is written in. The external DSML is normally parsed by the application using text parsing techniques. Considering the proposed solution described in chapter 1, the main application could be the web evaluation tool that is developed using a common programming language such as Java or C#. The external DSML would then be the HTML Document that needs to be parsed by the application in order to be validated. One of the advantages with using an external DSML is that its syntactic flexibility makes it easier for the domain expert to understand its contents.
- *Internal DSML*: is a particular way of using a general-purpose language so that only a subset of the language's features is used to define one small aspect of the overall system. Even though the DSML is defined in the applications language, it should still have the feel of a custom language. An example of this could be for instance an API.

3.2.1 Principles of Domain Specific Modelling Languages

There are several vital principles that must be followed in order to design an adequate DSML. First and foremost, the DSML must provide a decent level of abstraction that is intuitive to the developer and the domain expert. Secondly the DSML must be based on a common understanding of the domain, and not one-man's interpretation. Thirdly the DSML must evolve to satisfy the needs of the user or the context, otherwise it will quickly turn obsolete and become a

burden rather than an aid. Fourthly the DSML must be compatible with existing tools and methods in order to increase productivity within the domain. Finally the DSML should be designed in such a way that modifications to the DSML are implemented by extending the modelling language rather than modifying existing concepts (Open-Close Principle) [10]. In order to apply these principles to a modelling language, a developer requires a considerable amount of knowledge within the given domain, this is where the domain expert comes in contributing with the necessary expertise to design an adequate DSML.

Depending on the type of application the DSML is applied to, it is possible to classify DSMLs based on several characteristics [10]:

- *focus* - if it aims at a specific industry (vertical), or if it may be applied to multiple fields (horizontal).
- *style* - whether the language is declarative or imperative.
- *notation* - the language can be either graphical or textual.
- *internality* - The language is either specified in the host language (internal) or in a separate language (external).
- *execution* - Executing the model as a script at runtime, or that the model is used in a Model-to-Text transformation in order to generate something executable like source code.

3.2.2 Defining a Domain-Specific Modelling Language

Hopefully the previous sections have explained well enough what DSMLs are and why the solution presented in this paper should include them. This section contains an explanation on how to define a DSML and how it may be implemented into the solution.

A modelling language may be described as a textual or graphical representation of a concept which can be defined using three core ingredients; *the abstract syntax*, *the concrete syntax* and *the semantics* [10]. Figure 3.8 illustrates how the different ingredients relate to each other.

The Semantics defines the meaning of the elements included in the language in a formal manner.

The Abstract syntax describes the structure of the language and the way different primitives can be combined together. This is often represented as a meta-model which contains the definition for all valid models of the modelling language [10]. Considering the motivation explained in chapter 1 and the challenges presented in chapter 2, an abstract syntax for a DSML that attempts to describe a web page, could consist of a Meta-model representing how a valid HTML-document is structured, for instance that it consists of a tag-name, an identifier and a set of attributes. Any html element that deviates from this structure will not be considered as a valid html element. Furthermore the meta-model may also define how different HTML-elements are

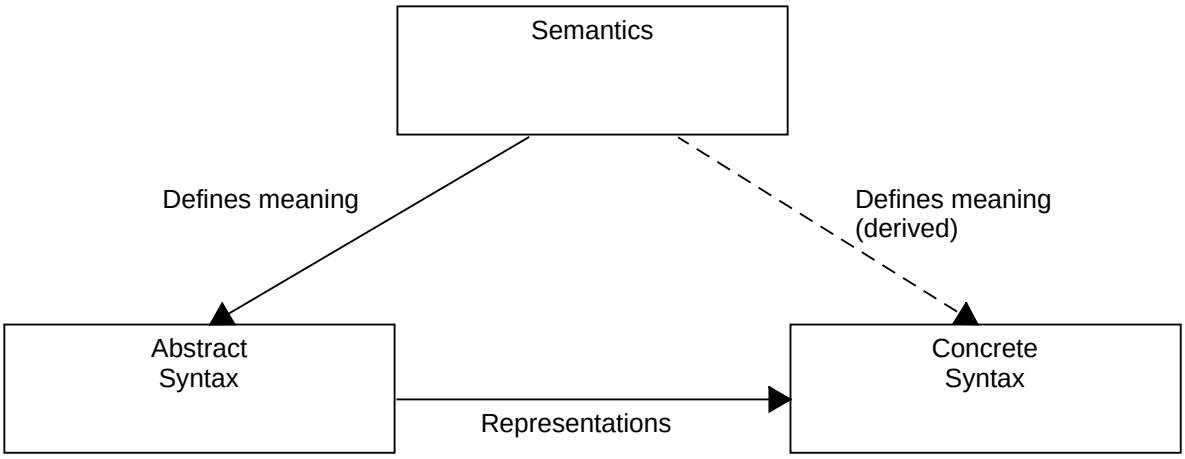


Figure 3.8: The three main ingredients of a modelling language

related to each other, such as the example presented in chapter 2 stating that every input element requires a label attached to it. Should the input-element lack such a relationship to a label-element, the model will again be considered as an invalid instance of the meta-model.

The development of the meta-model can be considered as the most important task when developing a DSML, since ultimately, it contains the rules for creating a valid model within the modelling language. The development of the meta-model can be broken down into 3 iterative and incremental steps [10]:

- **Modelling domain analysis**: The first step involves identifying several key aspects of the domain; the *purpose*, *realization* and *content* of the modelling language. One way of doing this is finding several reference examples within the domain. For instance by developing several sample web pages highlighting the criteria that must meet in order to be valid. One web page could consist of labelled input elements, another could contain the correct contrast ratio between the foreground elements and the background.

- **Modelling language design**: The second step involves using a meta-model to capture the abstract syntax based on the results from the previous step, thus creating a formalized meta-model for the modelling language that is being developed. The sample web pages created in the previous step would demonstrate how the HTML elements are structured and how they fit together.

- **Modelling language validation**: The final step involves instantiating the meta-model using modelled representations of the reference examples from the first step. This allows a domain expert to determine the completeness and correctness of the developed meta-model. Should the web pages be a valid instance of the meta-model that has been designed, it can bed claimed that the meta-model tested correctly, if not the meta-model has to be modified before repeating this step.

The Concrete syntax describes a textual or graphical representation of the language. The meta-model is not concerned with the concrete notation of the modelling language. Such a separation allows the abstract syntax to be mapped to multiple concrete syntax's regardless if they are textual or graphical. Considering the abstract syntax for a web page, it is possible to map the abstract syntax to a textual language such as an XSD and then check if the web page satisfies the constraints defined in it, alternatively it could map the abstract syntax to a graphical representation such as a UML model in a language workbench. Despite the fact that the concrete syntax's are presented very differently, they are both valid representations of the same modelling language.

When generating a graphical representation of the model language, there are several elements that must be defined: *graphical symbols* such as lines, shapes and textual information, *compositional rules* that define how the graphical symbols relate to each other, and *mapping* that bridges the graphical symbols to the abstract concepts in the modelling language, see figure 3.9 [10].

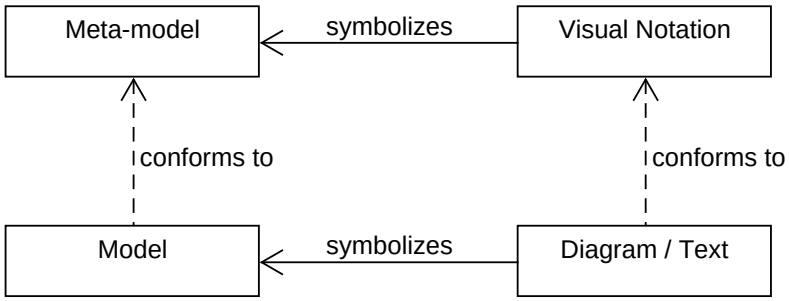


Figure 3.9: An illustration of a Graphical Concrete Syntax

As with the graphical representation, the textual representation of the modelling language maps the abstract context to the concrete syntax, only instead of mapping to figures it maps to textual elements such as pieces of code. There are several key elements when defining the textual concrete syntax: *Model information*, the textual syntax needs to be able to interpret the information stored in the abstract syntax, *Keywords*, words that hold special meaning in the textual language such as "int" and "String" that we find in Java, and *Scope borders* that are special symbols used to mark the beginning and the end of a specific section such as the curly bracket at the start and end of a method or class in Java. The textual concrete syntax also needs to define a list of *separation characters* that are used to split up elements in a list and *links* for connecting different elements.

Modelling language that have a clear and precise concrete syntax are able to take advantages of techniques such as automatic generation of tools for editing models defined within the DSML [10].

There are two scenarios we can be considered when implementing a DSML into the web evaluation tool: *the code first approach* where the DSML is to be implemented into an existing

application, and *the model first approach* where the DSML is to be implemented into an application that hasn't be developed.

In the code first approach the DSML is placed as a thin layer on top of the API in order to make it easier to manipulate the framework, this thin layer is referred to as *the semantic model* [28]. The DSML serves as a bridge between a conceptual modelling language within a specific domain and an existing framework that is to operate within that domain. The DSML makes it easier to understand how the framework operates within the domain [28].

In the model first approach the DSML is defined before the application is developed, which means that there is no API to base the modelling language on. With this approach it is common to identify a few scenarios and create an initial DSML based on them. Once the initial DSML is adequate, it can begin to implemented it into the framework, and continue to evolve by identifying more scenarios [28].

3.2.3 Parsing Domain-Specific Languages

A DSML that provides a clear and precise description of the domain is a good start, but in order for it to serve its purpose it must be applied to the system. This is achieved by parsing the DSML in order to obtain a *semantic model* which can be used in the system or one of it's components. Figure 3.10 shows an example of the DSML process.

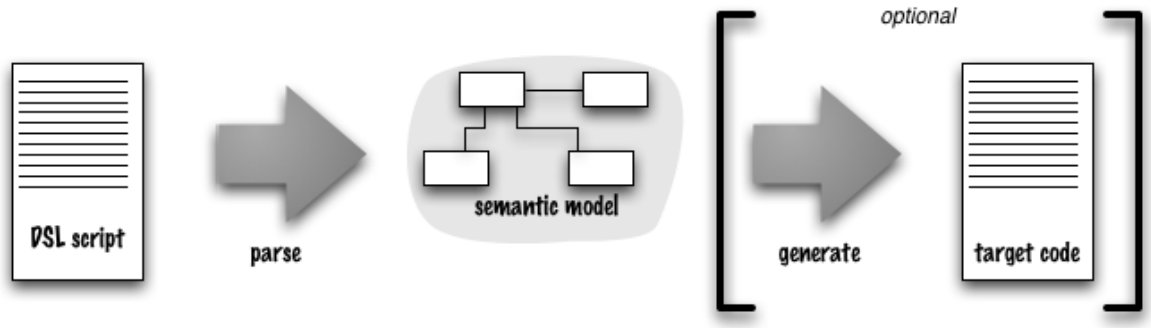


Figure 3.10: An example of the overall architecture of DSML processing [28]

The general idea is that all important semantic behaviour is captured in a model. The semantic model can be described as a model of the subject that the DSML describes. In other words a semantic model can be seen as a model that the DSML populates. In order for the semantic model to be of any use, it should be designed based on the purpose of the DSML. Basically it's a model that contains all the important semantic behaviour which is populated by a DSML via a parsing step. A meta-model could then be used to define the DSML that describes the semantic model [28].

The semantic model differs from the domain model even though both are quite similar. The Domain Model is often used to capture the core behaviour of an application, while the Semantic

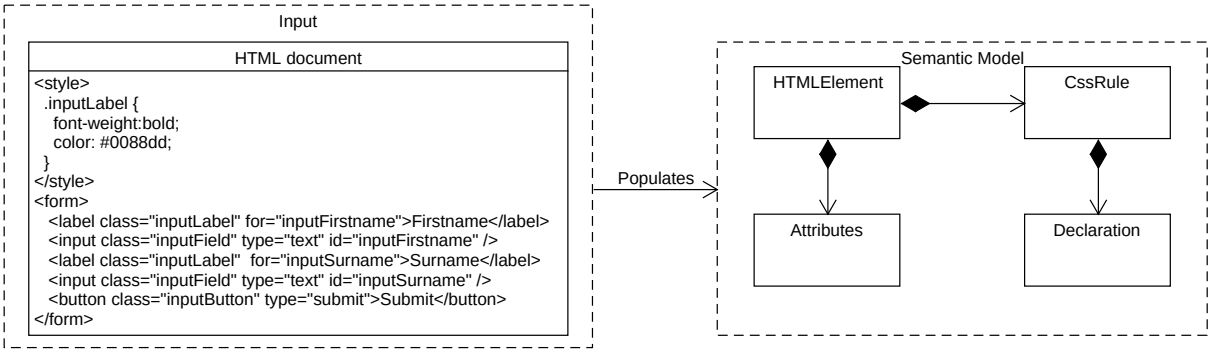


Figure 3.11: An illustration of a Semantic model being populated by a DSML script

model may consist of only the data [28].

Considering the requirements and the web pages that must satisfy these requirements presented in chapter 1, the semantic model could be a data structure that consists of html-elements, html-attributes, CSS-rules and CSS-declarations. The meta-model would then be a model of the DSML defining the types, how they relate to each other and the possibility of adding validations rules such as structural- and attached constraints. Modelling constraints will be explained in section 3.3.2.

The basic steps to the DSML parsing process consists of mapping a DSML text to a syntax tree which can then be used to populate the semantic model. Fortunately, html-elements are already structured as a tree and can therefore be easily mapped to the semantic model. The CSS-rules and -declarations are however slightly more tricky since it does not store it's information in any sort of usable data structure, but rather consists styling rules represented as key value pairs and grouped together by selectors. The Html and CSS domains will be explained in chapter 4. Figure 3.11 shows an example of a semantic model populated by a DSML script using a parsing process.

There are two parser processes that may be used to extract a syntax tree from a document: *the Delimiter-directed Translation* and *the Syntax-Directed Translation*. The remainder of this section will describe these two.

Delimiter-directed Translation

The Delimiter-directed translation is the simplest of the textual parsers commonly used. It works by basically reading in an input and breaking it down into smaller pieces based on a selected delimiter character, such as a line ending or semicolon. The advantage with this approach is how simple it is, but unfortunately it is ill suited for handling more complex scenarios, especially for languages that contain nested contexts. It is still able to parse complex languages, but must rely on pattern searching mechanisms such as *regular expressions* [52] to do so. Anyone who is familiar regular expression know that it has a very steep learning curve [28].

The Delimiter-directed Translation works well with simple languages, but becomes quickly messy and difficult to maintain when the language is complex. Unfortunately this type of parser is unsuitable even for the simplest of CSS-documents, since it won't be able to easily handle nesting in the CSS-document. See figure 3.12 for an illustration of the Delimiter-directed Translation process.

Syntax-Directed Translation

A computer language conforms to its grammar the same way a model conforms to it's meta-model. Just as the meta-model contains the definition of all its valid models, the grammar determines if the syntax of the computer language is legal or not. The Syntax-Directed Translation parser uses the language's grammar to create a parser that can map input text to a parse tree. The grammar makes this task possible by breaking the input text down into elements that can more easier be interpreted [28].

The Syntax-Directed Translation consists of two components: *the lexer* which splits the the input text into elements, and *the parser* that maps the elements to the syntax tree, then uses the syntax tree to populate the semantic model [28].

The Syntax-Directed Translation is not as simple as the Delimiter-directed Translation since it requires a well defined grammar for the language that needs to be parsed, but once it's defined it can handle much more complex languages than the Delimiter-directed Translation could ever hope to parse. See figure 3.12 for an illustration of the Syntax-Directed Translation process.

3.3 The Meta-Object Facility

One of the greatest challenges when sharing data between systems or integrating applications with each other is that the meta-data (the data about data) is often incompatible with other systems and can therefore not be shared outside the application. *The Meta-Object Facility (MOF)* is a standard defined by OMG that attempts to overcome this challenge by providing a platform-independent meta-data management framework and a set of meta-data services that grants distributing data and meta-data between systems that comply to MOF. In addition MOF also includes the concept of platform-independent meta-models, providing the means to map a MOF-based model to multiple platforms using model transformations such as MOF-to-Text, MOF-to-XML or MOF-to-Java [32].

MOF is based primarily on the Object-orientated modelling language and includes concepts such as classes with typed attributes and operations, and enables reuse by supporting multiple inheritance. It distinguishes between relationships that are fundamental, such as references to other classes, and those that are not fundamental which includes observations about the class [30]. For instance, a HTML input element referenced to by a HTML label element would be considered fundamental to that label element, where as the html input element being referenced to

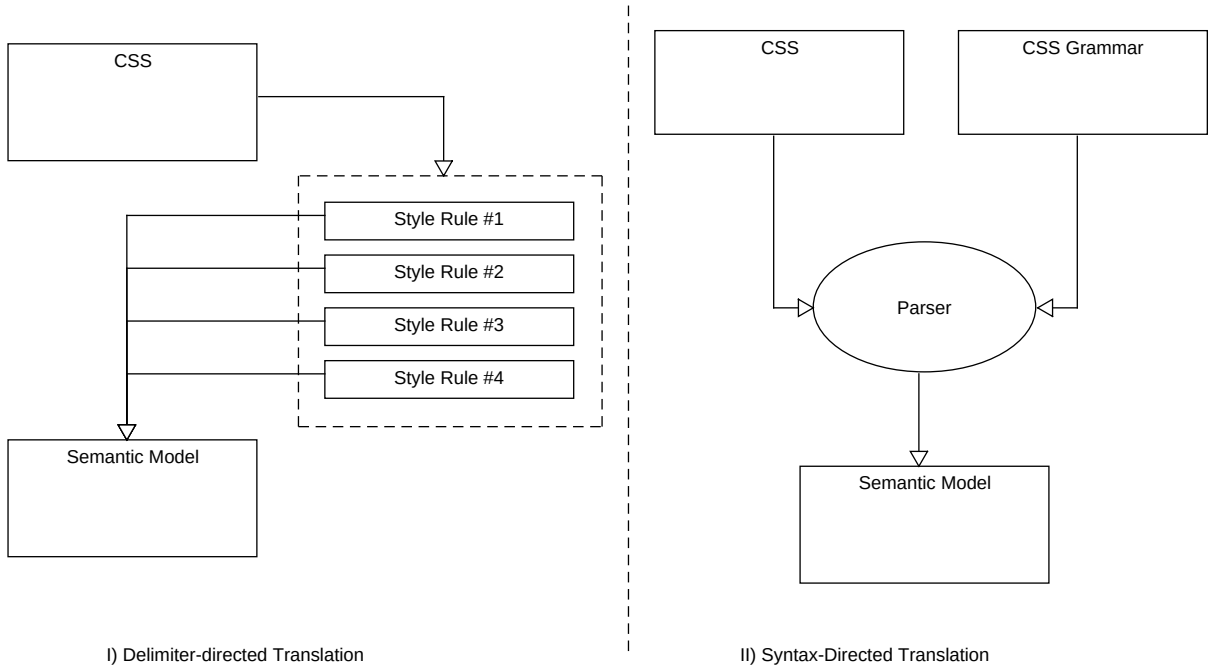


Figure 3.12: An example the Delimiter-directed Translation and the Syntax-Directed Translation [28]

would consider it's relationship to the label element as non-fundamental since it can be instantiated regardless if a label element references it or not.

In order to describe these relationships between classes, MOF allows objects to refer to each other by supporting the use of class-typed attributes. Expanding on the previous example, the label class would contain a reference of a type corresponding to the input class allowing the modeller to assign the reference with a descriptive name such as "labels" and the option to add modelling constraints. MOF can be considered as a specialized DSML for meta-modelling.

MOF supports complex containment rules in its meta model, allowing for instance classes to be nested within other classes to an unlimited depth [30]. This can be an advantage when attempting to model a web page since it can potentially contain an arbitrary number of html elements nested within each other.

Furthermore, MOF uses a common meta-model shared with UML, but extends it with the following MOF specific capabilities: *Reflection* which allows a model to describe itself, *Identifiers* which provides the ability to identify meta-model elements as unique objects and *Extension* that extends model elements with name / value pairs.[32]. Since the MOF model is reflective, it can be placed at the top level in the MDA hierarchy filling the role as the meta-meta-model.

In order to achieve it's primary goal of enabling data interchange between multiple systems, MOF uses the XMI standard to define how it's models and the instances of the models can be

exchanged using XML. This is done by using Document Type Definitions (DTD) or XML schemas generated from corresponding models [30].

3.3.1 The MOF model

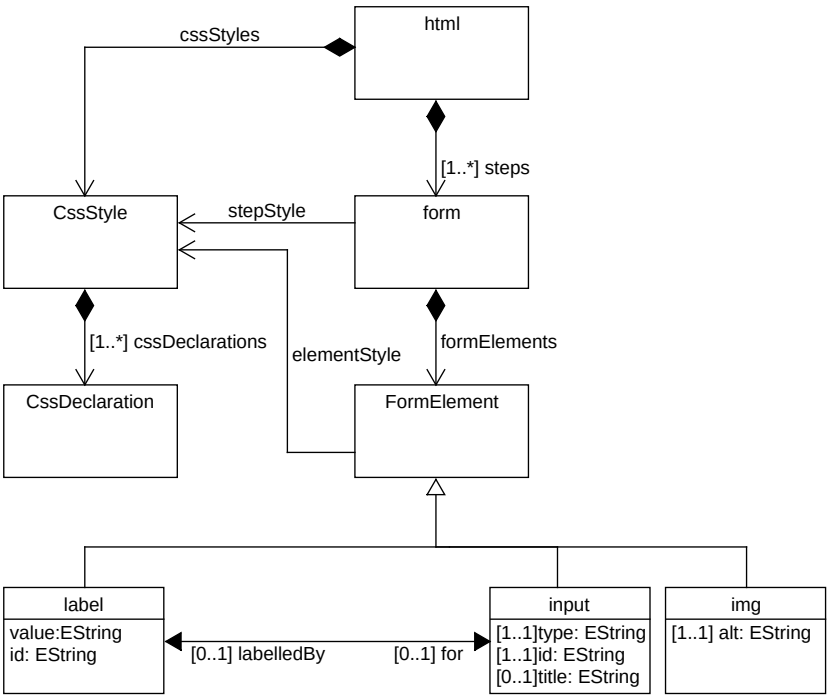


Figure 3.13: An example of a MOF-based model

As previously mentioned the MOF model defines the meta-meta-model at the top level in the meta-modelling hierarchy, see figure 3.3. The MOF model provides a common language that ties the meta-model and model together. Despite the MOF model being object-orientated, it is not limited to defining object-oriented meta-models, it can be used to describe other objects such as a web page [37].

Since the MOF model shares a common subset of the UML core, it's core modelling concepts corresponds (most of the time) with UMLs core modelling concepts. For instance, a MOF Class corresponds to a UML Class, a MOF attribute corresponds to a UML attribute and so on [37].

The MOF model basically consists of four main modelling concepts. *Classes*, which model MOF meta-objects, *Associations*, which model relationships between meta-objects, *DataTypes* which models primitive data, and finally *Packages* that modularizes the models [37].

Figure 3.13 illustrates an example of a MOF-based model of a web page consisting of one or more forms. Each form may contain an arbitrary number of Form Elements, that can either be a label, an input or an image. Every form and form element may refer to a CSS style that con-

sists of at least one styling rule or declaration. Finally the model includes several requirements to the form elements. Every image element must contain the *alt*-attribute, and every input element must contain an *ID*-attribute and a *type*-attribute. There may also not exist more than one relation between the label element and the input element. These requirements included in the MOF-based model are enforced using *model constraints*, which will be explained in the next section.

3.3.2 Model Constraints

In order for a model to be valid, it must conform to it's meta-model. In order to conform to it's meta-model, the model must be correctly typed and satisfy all of it's modelling constraints. This section will present MOF-based modelling constraints.

A modelling constraint is basically a restriction on parts of the instance of a model. It is possible to differentiate between two types of constraints in a MOF-based model: *structural constraints* and *attached constraints* [48]. Figure 3.14 demonstrates how structural constraints and attached constraints can be used to ensure that an input element has a relationship to a label element, and that the attribute *ratio* is equal to or greater than 7. The model $m1$ doesn't satisfy the structural constraints of the meta-model since it's missing the bidirectional relation between the label element and input element. The model $m2$ fails to satisfy the attach constraint of the meta-model since the value of the *ratio*-attribute less than 7.

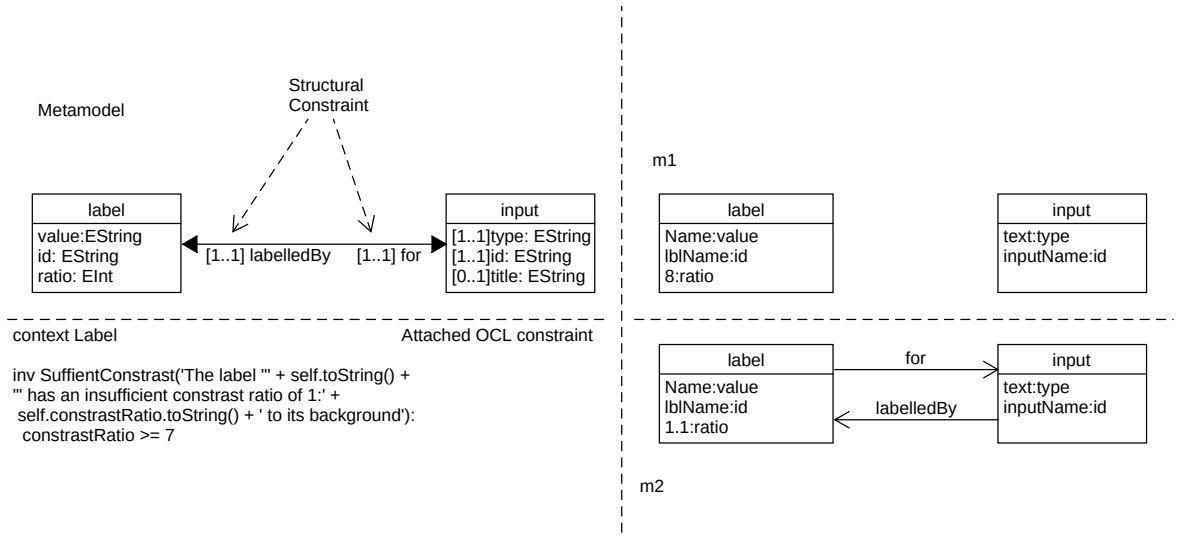


Figure 3.14: An example of structural constraints and attached constraints.

Structural Constraints

Structural constraints are basic restrictions that are applied to properties of classes in the meta-model to the model in a MOF-based language. They are generally limited to simple constraints such as cardinality and uniqueness. Additionally structural constraints include typing constraints, that limits which types of elements the model can contain and how these elements are related to each other. Typing constraints are also defined in the meta-model [48].

Attached Constraints

Structural constraints tend to prove insufficient when attempting to specify more complex system requirements. In order to define these complex constraints it is possible to use a textual constraint language such as *the Object Constraint Language (OCL)*. These sorts of textual constraints are commonly referred as attached constraints. Attached constraints are normally used to define requirements that are not possible to represent with structural constraints. [48]

OCL is a formal constraint language that is used to define restrictions on UML models. These restrictions are described as conditions that the model must uphold. One of the advantages with OCL is that it's purely a query language, and therefore doesn't alter the model in any way, avoiding unpredicted side-effect. Furthermore the OCL is a typed language, this provides the means to check if the expression is well formed, thus preventing problems such as ambiguity in the modelling constraint [22].

Since the OCL is part of the UML specifications, it's semantics are based on UML concepts and constructs. This means that certain concepts found in OCL might not correspond to matching concepts in MOF, and depending on how it is implemented, might require some parts of the OCL specification to be transformed in order to be used in a MOF-based modelling constraint [22].

3.4 The Diagram Predicate Framework

The Diagram Predicate Framework (DPF) is an ongoing research project between the Bergen University College and the University of Bergen. The goal of the project is to introduce a formal approach to domain-specific languages, meta-modelling, and model transformations using a graph-based specification framework based on category theory. The framework consists of a set of meta-models arranged into hierarchy where each meta-model at a specific level conforms to it's adjacent meta-model one level above. The meta-models are formalized at any level as diagrammatic specifications. DPF uses graph homomorphism when checking for type conformance between a model and it's corresponding meta-model. [48].

One of the problems we face when defining Domain-specific modelling languages is that

normally DSMLs are specified by a graph-based meta-model combined with text-based constraints, this means that the typing and the constraints are specified in different languages and model transformations are usually not constraint-aware. This is not a problem with DPF since the formalization introduced by the framework considers both the typing and the constraints by supporting first-order logic and categorical logic [48].

As explained in the previous section, the MOF-model shares a common meta-model with UML and therefore includes many similar concepts and features that it primarily tailored for modelling object oriented systems. In it's most abstract form, a DPF-model consists only of an arrow and a node, see figure 3.15. This makes DPF a much more generic framework which can be used to model more or less anything without including irrelevant elements or concepts which would be included when using MOF.

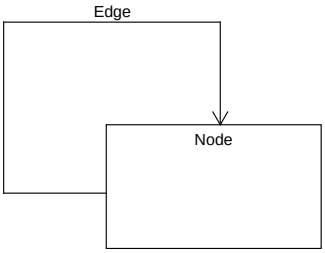


Figure 3.15: Top level model in the DPF metamodel hierarchy

Specifications

DPF refers to models at any level in the meta-model hierarchy as diagrammatic specifications. Each specification consists of an underlying graph combined with a set of atomic constraints. These atomic constraints are specified by predicates from a predefined *signature*, and represent properties which instances of the specification must satisfy. We define the semantics of a specification by the set of its instances, an instance is a graph combined with a graph homomorphism that satisfies the specified atomic constraints. DPF uses graph homomorphism to formalize the conformance relation between an instance and it's specification [48].

Figure 3.16 shows an example of a specification consisting of an underlying graph containing two nodes and an arrowing, and a signature with a predicate defining an atomic constraint for multiplicity.

Signatures

As mentioned in the previous section, DPF handles constraints in its graphs, and not as an additional textual pieces of code like the MOF-model does. This is achieved using diagrammatic predicates. A predicate consists of a name, a shape, a visualization and a semantic interpretation. A signature consists of a collection of predicates [48].

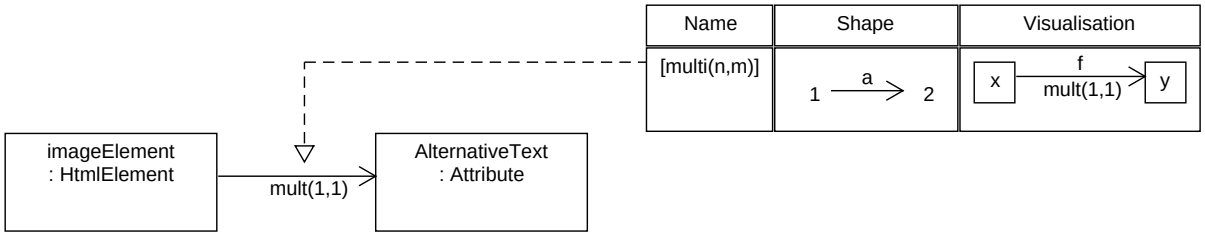


Figure 3.16: An example of a specification with it's corresponding signature

Table 3.1 contains several examples of predicates we might find in a signature. The modelling environment decides what kinds of predicates are included in the signature.

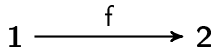
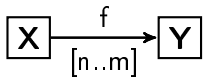
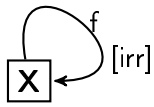
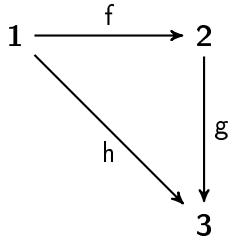
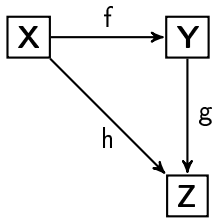
Name	**Shape**	**Visualization**	**Semantic Interpretation**
[multi(n,m)]			$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[irreflexive]			$\forall x \in X : x \notin f(x)$
[composition]			$\forall x \in X : h(x) = \cup\{g(y) \lvert y \in f(x)\}$

Table 3.1: Examples of predicates in a signature

There are several fundamental problems with using attached constraints like OCL constraints in a diagrammatic model. First of all, the validation must consist of at least two steps, first the model typing and structural constraints must be checked, then the OCL constraints. Second of all, any changes to the model must be reflected in the any OCL constraint related to the model, this makes maintenance of the model more challenging and tedious. Third of all, in order to fulfil the MDE's vision of *"everything is a model"*, the structure and the constraints should be defined in the same format. Finally, by using attached constraints parts of the semantics of the model is hidden, since generally these constraints are not visible, and even if they were a domain-expert will have difficulties understanding the OCL constraint [48].

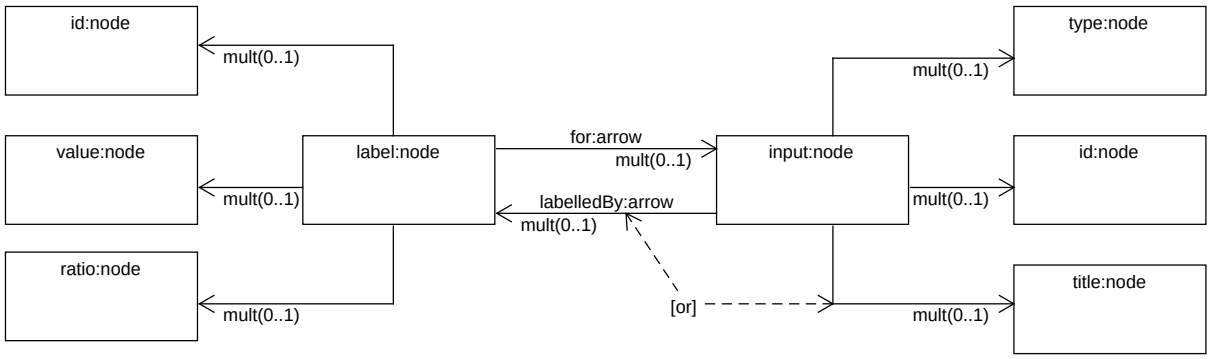


Figure 3.17: An example of a metamodel defined using DPF

3.4.1 Metamodelling in DPF

As mentioned in section 3.2, meta-models are used to define the abstract syntax of a modelling language. Models are considered to be syntactically correct if they conform to the meta-model that defines the modelling language. DPF has a similar approach to specifications adjacent each other in the meta-modelling hierarchy. A specification is considered correct if it conforms to the specification one level above [48]. Figure 3.18 expands upon figure 3.3 illustrating how a the meta-model hierarchy looks in DPF and how it compares to the more traditional meta-model hierarchy defined by OMG.

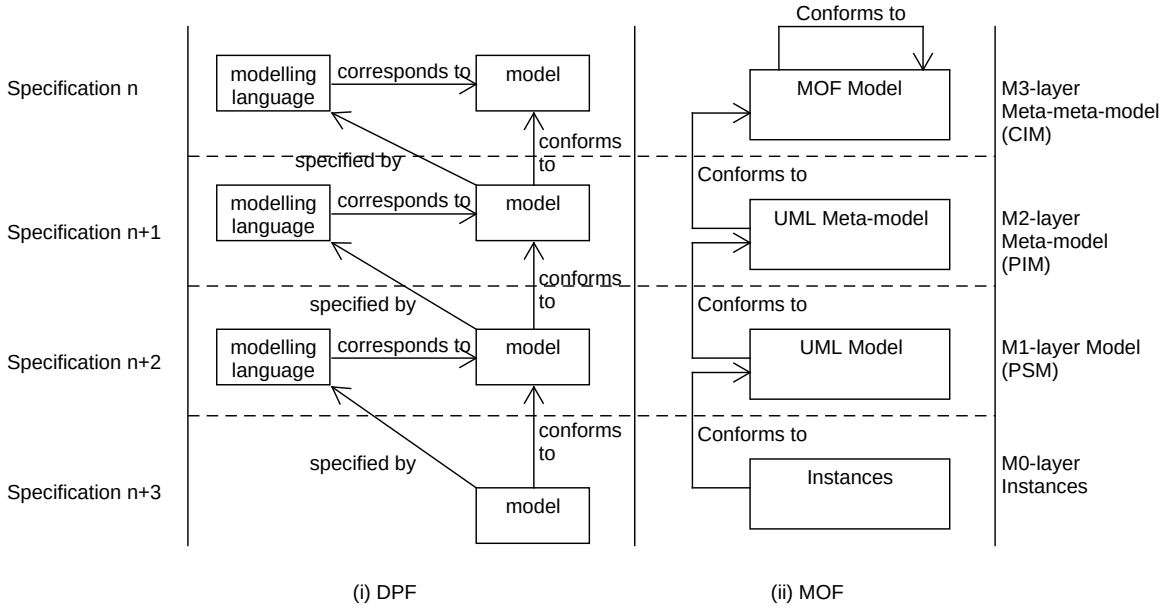


Figure 3.18: A comparison of the MOF-based meta-model hierachy and the DPF meta-model hierarchy [4]

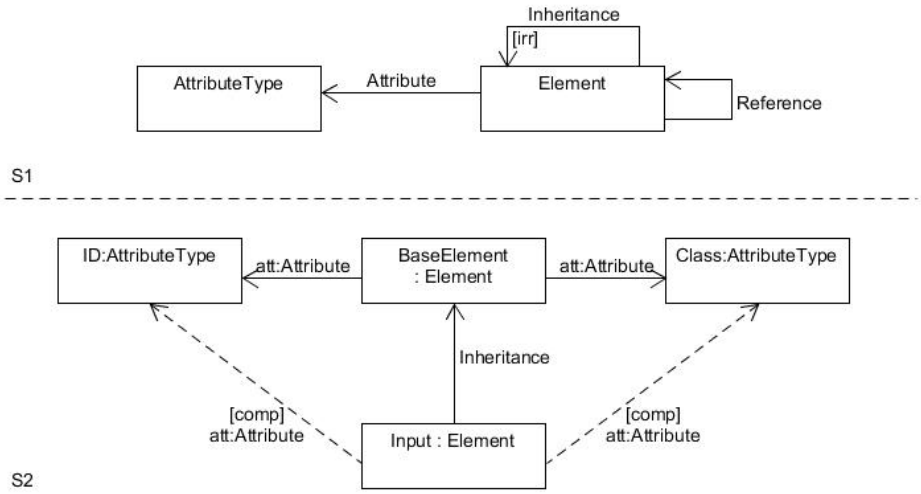


Figure 3.19: An illustration of inheritance in DPF

In theory, the meta-modelling hierarchy defined using DPF (and actually MOF as well), may consist of an infinite number of levels. Generally though a reflexive meta-model serves as a natural top model for the hierarchy. In DPF the generic model consisting of a node and an arrow is used (see figure 3.15), in MOF the MOF-model is used since it has the necessary reflexive trait.

DPF distinguishes between two types of conformance: *typed by* and *conforms to*. A model is considered to be correctly typed if there exists a typing morphism between the model and it's meta-model. A model conforms to it's meta-model if it is correctly typed and satisfies the constraints specified on the meta-model [48].

As mentioned in the introduction to this section, DPF is a generic framework meaning it is not limited to modelling object oriented structural models. A reasonable question is if DPF can adequately model an object oriented software system, and if it can so as well as other modelling frameworks such as MOF. Two of the more complex aspects when modelling an object oriented structural model is *inheritance* and *containment*. These features are simple to model in a MOF-based model since it is closely related to UML.

In DPF inheritance between two classes can be represented by using an arrow specified as an inheritance arrow. When class X' inherits from class X it gains access to all of it's attributes and references by composing the inheritance arrow with the attributes and references arrows. An inheritance relationship between two classes can however not be circular, if class X' inherits from class X, class X may not inherit from X'. This can be enforced by using the irreflexive predicate [48].

Although html elements can not inherit from each other, being able to model inheritance at a higher level of abstraction makes the overall DPF model easier to interpret. For instance, almost all HTML-elements support the attribute ID and class, it is possible to dramatically reduce the number of nodes in a specification by modelling all input elements to inherit from a generic

HTML-element node. Figure 3.19 shows how inheritance in DPF can be used to model HTML-elements.

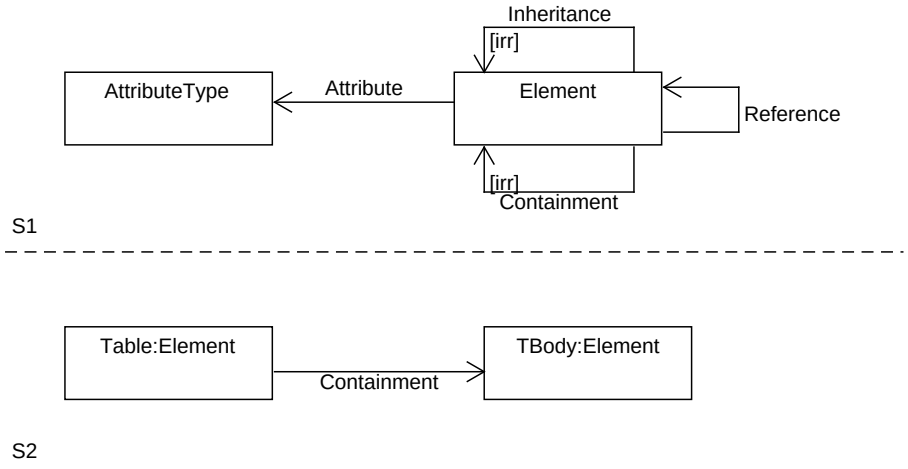


Figure 3.20: An illustration of containment in DPF

When it comes to modelling the containment relationship in DPF, it is pretty straight forward. When modelling a containment where class X contains class Y, an instance of class Y becomes dependent on an instance of class X. This means that instances of class Y only exist if related to an instance of class X. In DPF the containment relation is represented by a containment arrow.

Figure 3.20 demonstrates how containment in DPF can be used to model the relation between a table-element and a tbody-element. The tbody-element may only exist within the table element tag, using the containment relation it is possible to specify this.

3.5 Summary

This chapter has explained several different background theories which will serve as the foundation for the solution proposed in this paper. The general idea is that by defining a well formed domain-specific modelling language, it should be entirely possible to create a model representation of many the requirements described in chapter 1 using a modelling framework such as MOF or DPF. This model will serve as the core of the solution proposed in this paper and attempt to overcome the challenges presented in chapter 2.

For starters, the meta-model that the model conforms to, will ensure that the model itself is defined in a clear and precise way, thus preventing problems that might occur because of ambiguous requirements or misunderstandings between the developer and the domain expert. Furthermore, as explained in section 3.1.2, models can easily be represented at different levels of abstraction including both platform specific models and platform independent models.

Chapter 4 will attempt to define a modelling language that can adequately describe a model representation of the WCAG 2.0 requirements.

Moreover, section 3.1.1 explains that models can be used for much more than providing documentation for a system. Models can be used by a number of MDSE techniques such as transformation, validation, analyses, execution and so on. Using these techniques it might be possible to provide insight into the evaluation process, thus contributing to a solution for the transparency problem. Chapter 5 will discuss how a model can be used to evaluate a web page.

Chapter 4

Meta-modelling

This chapter will begin with an explanation of the web page domain, followed by a short discussion justifying the choice of modelling framework. The DSML will then be formalized using the chosen modelling framework, with a detailed description of which requirements it is attempting to model. This chapter will end with a conclusion where the model will be evaluated based on how many of the requirements it was able to model, and if the model may solve any of the challenges presented in chapter 2.

The final solution should include a DSML that contains all html elements supported by HTML5 [33] and HTML 4.01, however the DSML presented in this thesis will be limited to the HTML elements that are included in the Universal Design requirements presented in chapter one. Another limitation to the DSML in this thesis, is that it will only describe html elements and attributes that are supported in HTML5, ideally the final solution would also include elements and attributes that are supported in HTML 4.01 and XHTML 1.0.

4.1 Describing the Domain

Considering the challenges presented in chapter 2, it would be a natural assumption that the DSML will focus solely on the HTML domain, however the structure of the html elements that make up the web page is only part of the problem. In order to ensure that the web page is usable by users with impaired sight, the presentation of the web page must also be considered. Thus the domain that needs to be modelled is more than just the html structure, but the entire web page including it's styling defined by *Cascading Style Sheets (CSS)*. The web page domain can be considered as a container for the two sub-domains that should be included in the DSML: *The HTML domain* and *the CSS domain*. One could argue that a *Script domain* defining the javascript elements within the web page also belongs in the web page domain, but such a domain is outside for the scope of this thesis and will therefore not be included in the solution. Figure 4.1 illustrates the how this thesis views the web page domain.

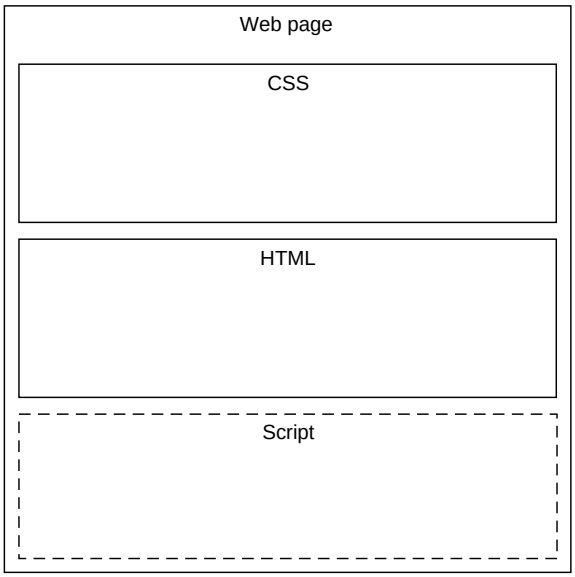


Figure 4.1: The Web page domain and it's three sub-domains: HTML, CSS and Script

This section will give a short technical description of the two sub-domains, in order to provide some context for the next section where the WCAG 2.0 model will formally be defined.

4.1.1 Hyper Text Markup Language (HTML)

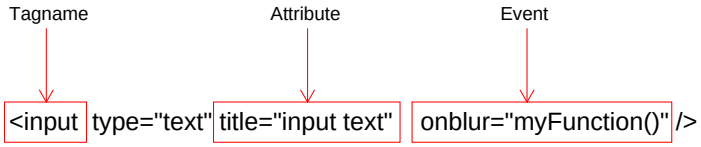


Figure 4.2: An example of the a html input element

A HTML document basically consists of a tree of elements and texts, every element in the tree consist of a start tag and an end tag, which can be nested within other elements in an identical manner to a XML document. Each element may contain an arbitrary number of attributes which determines the behaviour of the element. An attribute consists of a name and a value separated by the "="-character. HTML user agents such as Web browsers parse the HTML document transforming it into an in-memory tree representation of the document, commonly referred to as the *Document Object Model* (DOM) tree [33]. Figure 4.2 demonstrates how an html element is correctly written.

The DOM tree consists of several types of nodes: a *DocumentType* node, *Element* nodes, *Text* nodes and *Comment* nodes [33]. However, it's only the *Element* nodes that are of interest

when considering the requirements that are to be modelled in the next section. Each object in the DOM tree contains an interface that can be accessed by for instance javascript, this is also however outside the scope of this thesis.

One of the advantages with representing interactive content as a HTML document, is that the content becomes media-independent in that it can be rendered to multiple outputs such as screens, speech synthesizers and braille displays.

4.1.2 Cascading Styling Sheet (CSS)

A cascading styling sheet consists of a set of css rules, each rule has two parts: a *selector* that binds the style to a html element, and a *declaration* which defines the presentation of the selected html element. Furthermore, the declaration also consists of two parts: a *property name* and a *property value* [8].

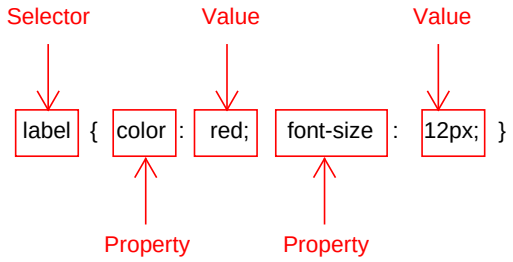


Figure 4.3: An example of a CSS rule

CSS rules in a style sheet are unfortunately not organized into a tree structure, like the Html elements are. Furthermore, CSS rules can be nested in other CSS rules. This doesn't grant any kind of inheritance, but rather allows the use of *Descendant selectors*. A descendant selector is basically a selector applied to certain elements within another element. For instance a descendant selector could apply styling to all labels within a given form.

Certain declarations in a CSS rule can inherit values from other corresponding declarations in other CSS rules. It isn't however the relationship between two CSS rules in a style sheet (i.e. nesting) that determines the value of the inherited declaration, but the relationship between the Html Elements these CSS rules are attached to. For instance, if a CSS rule containing a declaration for styling all fonts the colour blue is attached to a html form element, all label elements within the form element can inherit that colour value and thus render there contents with the font colour blue. A declaration will only attempt to inherit a property from another CSS rule if it contains a property assigned with the value *inherit*.

4.2 Choosing a Modelling Framework

One of the most important aspects that must be considered when selecting a framework is understanding what the domain is and how it is structured. Had the domain been a software system consisting of classes defined in an Object-oriented language, the Meta-Object framework would clearly be preferable since it specialises in defining DSMLs for object-oriented modelling languages.

However the domain in question is not defined with an object oriented language, it does not contain methods and classes in the same manner as one would find in Java or C#. As mentioned in the chapter 3, the MOF model isn't necessarily limited to only modelling object oriented systems and can in fact be used to model large variety of domains including web pages, but this includes a lot of overhead such as[37][50]: packages, imports, primitives, etc, which isn't of interest when modelling the web page domain described in section 4.1. DPF however doesn't contain such an elaborate meta-model at its core, but rather a simple meta-model consisting of a node and a reflexive directed edge pointing to the node, as shown in figure 3.15. This makes DPF a more generic framework when compared with MOF, and as such it is possible to model practically anything and without the danger of including concepts that are intended for other types of domains.

In addition to including unnecessary concepts, there is another challenge with modelling a web page with MOF. Some of the concepts that are included in the MOF model share the same notation as some of the concepts that are found the web page domain such as *class* and *attribute*. Using these common keywords could lead to ambiguity problems when modelling.

One last factor that should be included when deciding between MOF and DPF is how the attached constraints are included in the model. In order to handle challenges such as the transparency problem, the correctness problem or the completion problem, it is vital that the domain expert is able to understand the entire model. Including attached constraints in a graph-based representation is vital when determining if the model correctly represents the domain, since a constraint can ultimately decide if an instance of the model is valid or not.

In chapter 3 it was explained that an attached constraint is defined by attaching an OCL statement to the model. OCL is a purely textual language and therefore does not contain a visual representation. It is however possible to work around this problem by including a comment in the model that could serve as a representation of the OCL statement, but this isn't an optimal solution since changes to the comment (i.e. removing or moving the comment) does not change the way the OCL statement influences the model. DPF defines it's attached constraints as predicates which contain a visual notation that is included in the graph-based representation of the model, notifying the domain expert that there is a constraint attached to the corresponding edge.

The model designed in this chapter must be well-formed and provide a sufficient understanding to all parties that intend to use it. Furthermore being a DSML, it's expressive capabilities must be limited to only describing the web page domain in order to avoid ambiguity

problems. Taking into account the aspects mentioned in this section, this thesis will use DPF to model the web page domain.

4.3 The WCAG 2.0 model

As mentioned in the summary of chapter 3, one of the primary functions of our meta-model is to provide the developer and the domain expert a common understanding of the selected requirements, thus preventing problems caused by ambiguity. The model described in this section consists of two specifications: $\mathfrak{S}_0$ and $\mathfrak{S}_1$. $\mathfrak{S}_0$ functions as the meta-model to $\mathfrak{S}_1$, the arbitrary web page that is to validated must be correctly typed and satisfy the constraints defined in $\mathfrak{S}_1$.

4.3.1 Specification $\mathfrak{S}_0$

The specification $\mathfrak{S}_0$ specifies the most abstract view of a web page element. Every element may contain an arbitrary number of predefined attributes and events. Even though the Universal Design requirements selected in chapter 2 does not include any criteria for how an element's events should be implemented, it is still included in the $\mathfrak{S}_1$ specification in order to provide correct typing. Figure 4.4 displays the different common components of a web page element.

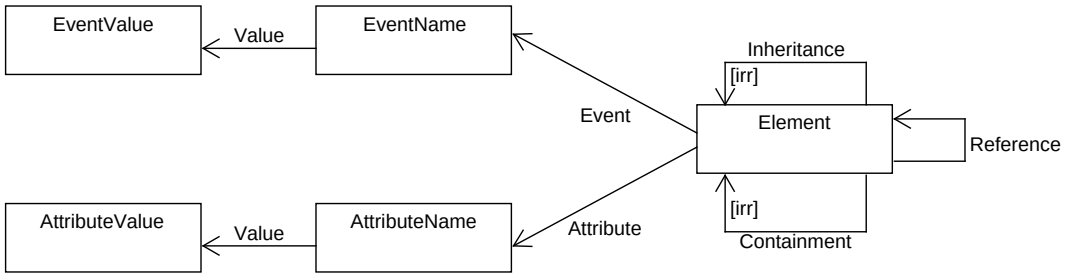


Figure 4.4: Specification $\mathfrak{S}_0$

The relationship between any two web page elements can boil down to two types. The first type is by reference, where one element refers to another element by using an identifier such as the element's ID. An example of this is for instance a label element referencing an input element using the *for*-attribute. This is represented in the $\mathfrak{S}_0$ specification with the *Reference arrow*. The second type of relationship is by containment, where one element serves as a container to another element. Since the HTML document is structured as a tree, every element (apart from the root element) is contained by another element. This is represented using the *Containment arrow*. The *irreflexive predicate* [irr] is added to the containment arrow to prevent an element from containing itself.

Specification $\mathfrak{S}_0$ includes a third relationship between elements in the WCAG 2.0 model, the

inheritance relationship is represented using the *inheritance arrow*. Even though as mentioned in chapter 3, HTML elements are not able to inherit attributes and events from one another, supporting inheritance at a more abstract view of a web page improves the model's readability. In order to avoid a circular reference, the *irreflexive predicate* [irr] is added to the inheritance arrow.

4.3.2 Specification $\mathfrak{S}_1$

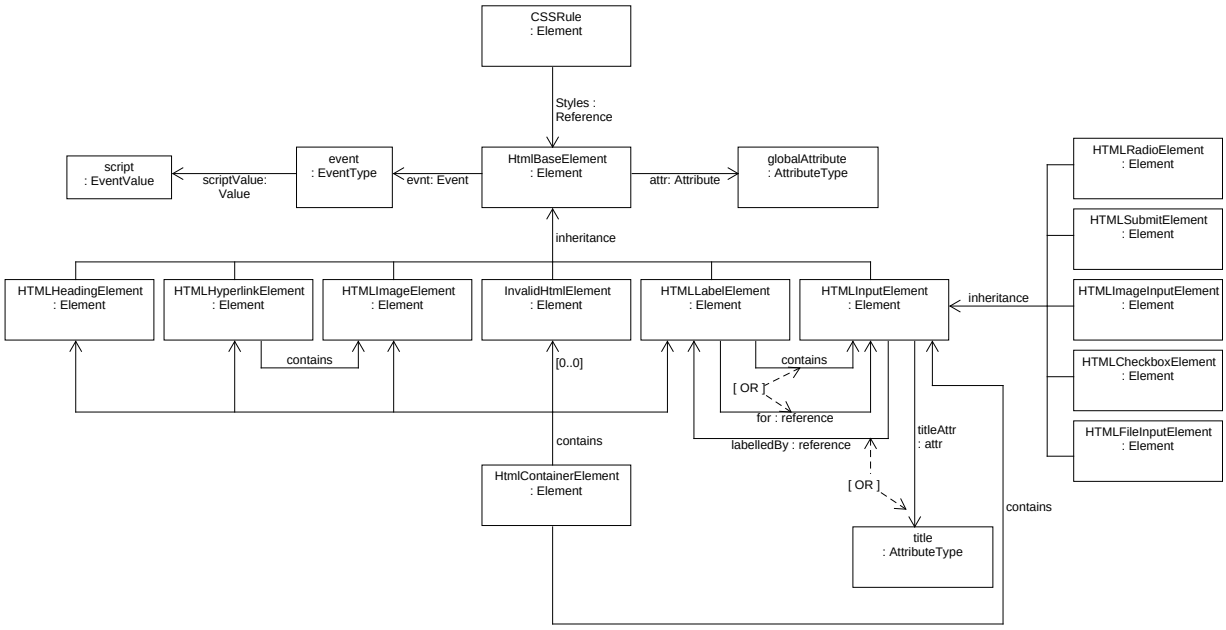


Figure 4.5: Specification $\mathfrak{S}_1$

In order to keep specification $\mathfrak{S}_1$ organized and readable, figure 4.5 will not include the various attributes belonging to each html element included in the specification. These attributes will be included with their own corresponding html element in separate figures that are either included in this chapter or can be found in appendix B.

One basic rule that seems to be at the core of the WAG 2.0 requirements is that developer must not use elements, attributes or attribute values for purposes other than what they're intended for. Deviating from correct use might prevent the web page from being rendered correctly by the browser or electronic aids [56].

The first thing that should be mentioned about the HTML domain is that every html element that belongs to the domain share a common set of attributes and events [33]. This is specified in the *HtmlBaseElement* element which is shown in figure 4.6. Every element with the exception of *CSSRule* inherits from the *HtmlBaseElement* element. The *HtmlBaseElement* includes a generic *EventType* that will capture all html events, and a *GlobalAttribute* container that consists of every

valid attribute a html element may have. Note that apart from the *data-** attribute type, there may not exist duplicate attributes in a *HtmlBaseElement*.

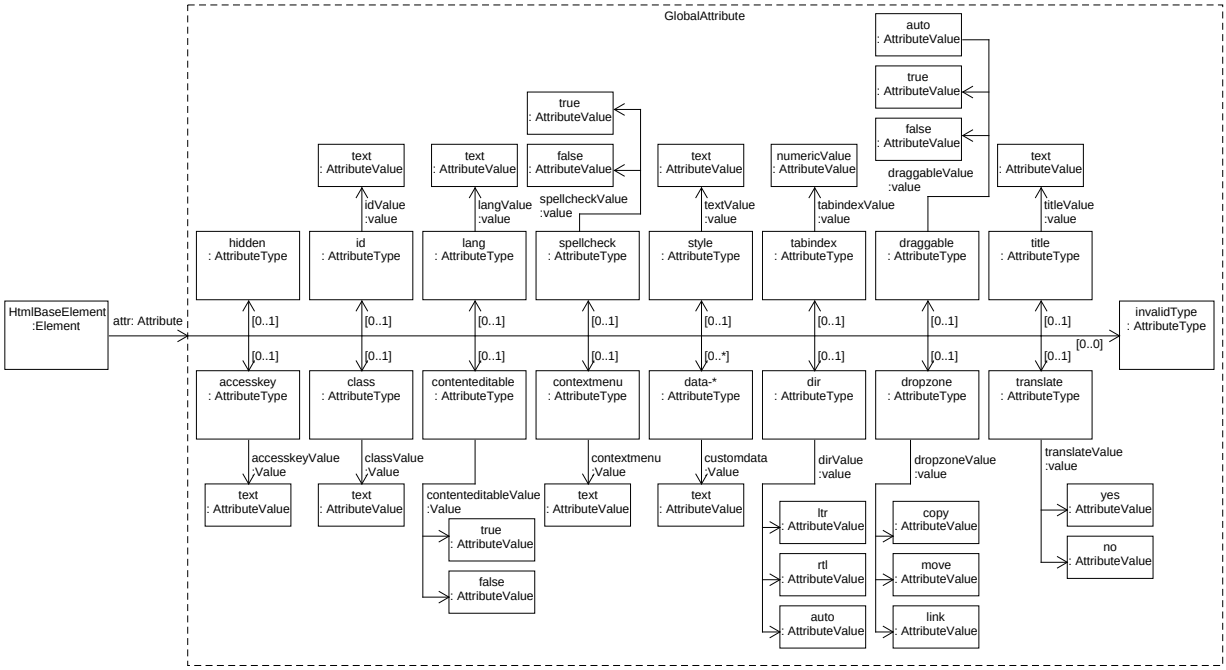


Figure 4.6: The Html Element Specification

The second thing that should be noticed, is the inheritance relationship between the *HTMLInputElement* and several other Elements. The *HTMLInputElement* represents a set of different inputs determined by the *type* attribute. Most of these input elements have the same attributes, but there are some exceptions. These exceptions are defined as separate elements that inherit the common attributes contained in the *HtmlInputElement* element, and extend it with their input type specific attributes. Since the inheritance relation is transitive, the input elements that are defined separately may still access the attributes and events in the *HtmlElement* element. Figure 4.8 displays the *HtmlInputElement* specification, and figures B.6, B.5, B.4, B.7 and B.8 in the appendix, show the elements that inherits from *HtmlInputElement*.

As specified in the previous section about $\mathfrak{S}_0$, html elements are organized into a tree structure, which means a html element will normally be contained within another html element. However, only certain html elements can function as containers for other html elements, these elements are defined as a *HtmlContainerElement*, see figure 4.7. Additionally label elements may be attached to input elements by containing them, and html anchor elements can contain image elements. These relations are modelled in figure 4.5.

Finally, a web page should only contain valid Html elements. With the rise of client-side frameworks such as AngularJS [16], it is not unusual for Html Documents to contain custom Html elements that will not necessarily conform to the HTML specification [57]. This rule is enforced by including the node *InvalidHtmlElement* and a constraint on the *contains* edge going from the *HtmlContainerElement* node to the *InvalidHtmlElement* node. This means that should

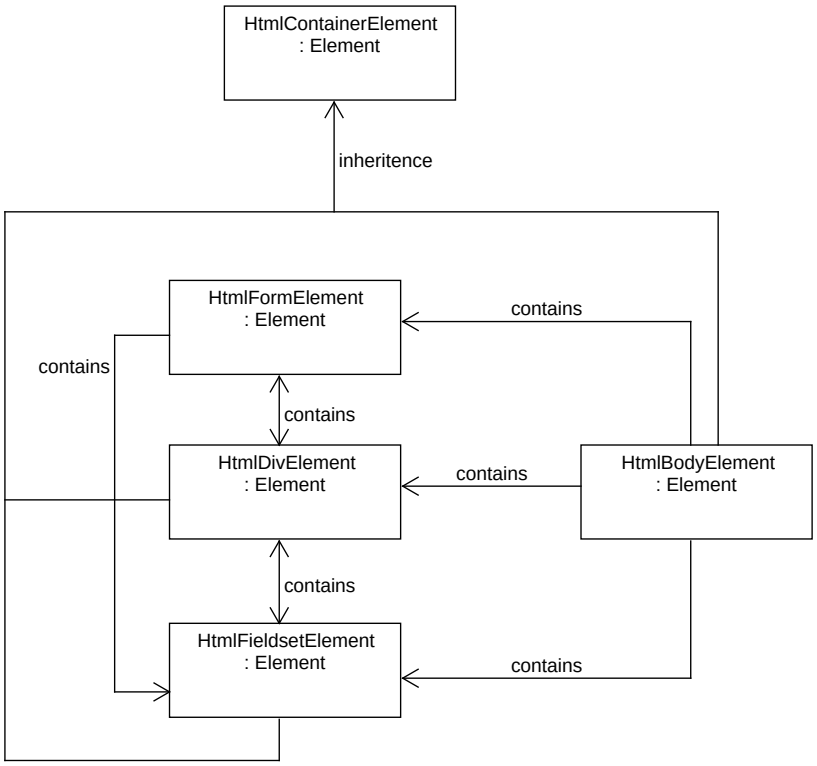


Figure 4.7: The Container Element Specification

any *HtmlContainerElement* contain a node of type *InvalidHtmlElement*, that node should be rendered as invalid. An identical constraint is added in the *HtmlBaseElement*, where an *invalid-Type* node has be added with a multiplicity constraint enforcing that no such node is permitted. See figure 4.6.

The remainder of this section will focus on the requirements that must be fulfilled and how they may be implemented into specification $\mathfrak{S}_1$. The requirements will be categorized into three groups: *Images and graphic requirements*, *Text and structure requirements* and *contrast and styling requirements*.

Images and graphic requirements

1.1.1 Non-text Content [56] requires all non-textual content that is presented to the user contains an alternative text that achieves the equivalent purpose. For instance, if the web page in question contains an image of a red car, an alternative text should be included stating that the image contains a red car. The purpose of this requirement is to ensure that users that are not able to view the image clearly (or at all), either due to reduced vision or some technological limitation, are able to understand the contents of the image using the alternative text.

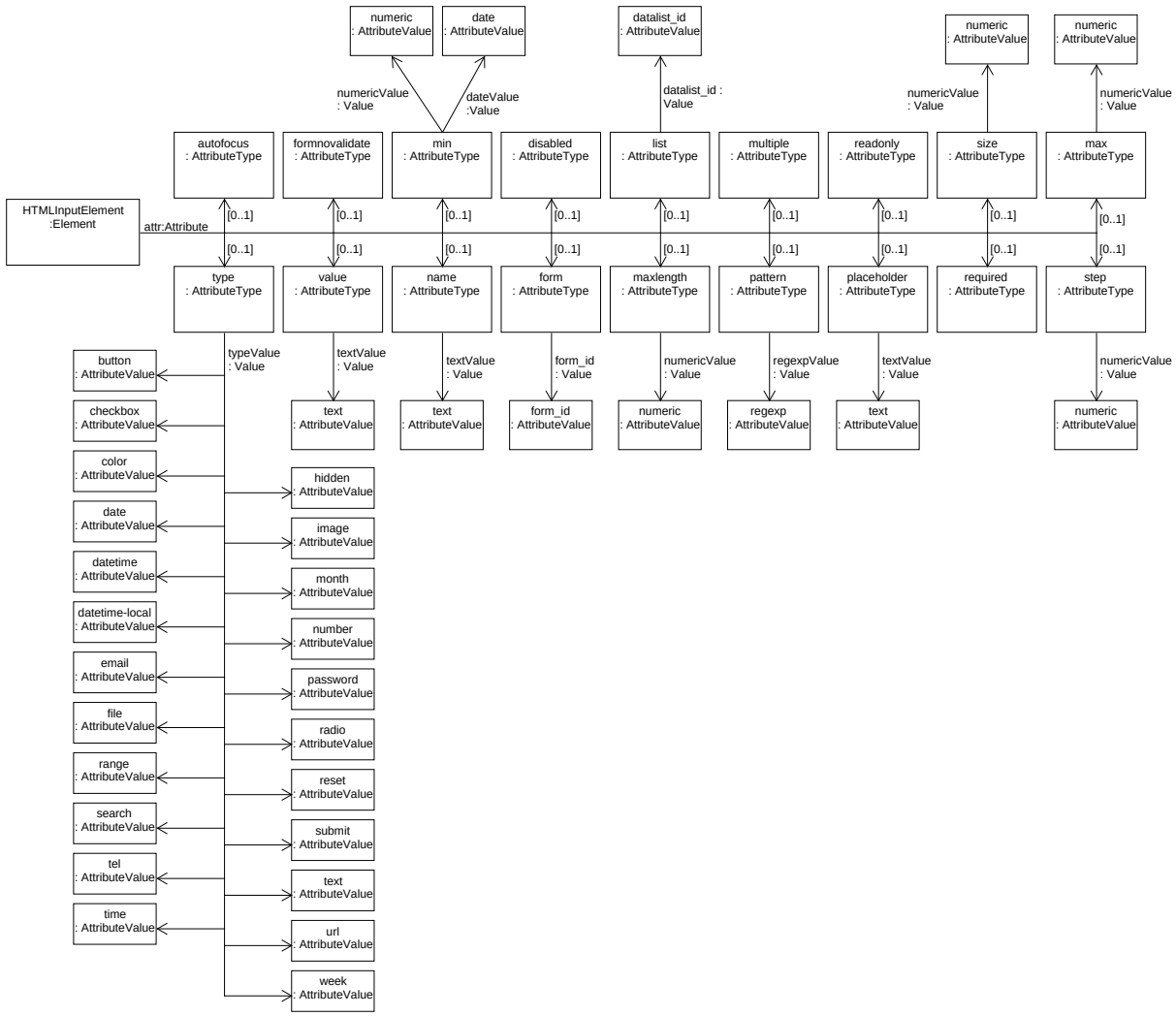


Figure 4.8: The Input Element Specification

1.4.5 Images of Text [56] requires that all visual elements, such as images, that are displayed in the browser should use textual content to relay it's contents instead of relying on the text in the element, for instance text in an image. An exception to this rule is situations where it is essential to include text in the image such as logos or brand names.

1.4.9 Images of Text (No Exception) [56] is an optional requirement that recommends all images containing text should be used for the sole purpose of providing decoration to the web page, and not attempt to communicate any kind of information to the user.

Without using some kind of advanced image processing tool, there is no way to determine programmatically if an image contains textual content or not. It is also difficult to determine if the image element contains a satisfactory description of the image. We can however ensure that there exists a description for the image and that it is properly attached.

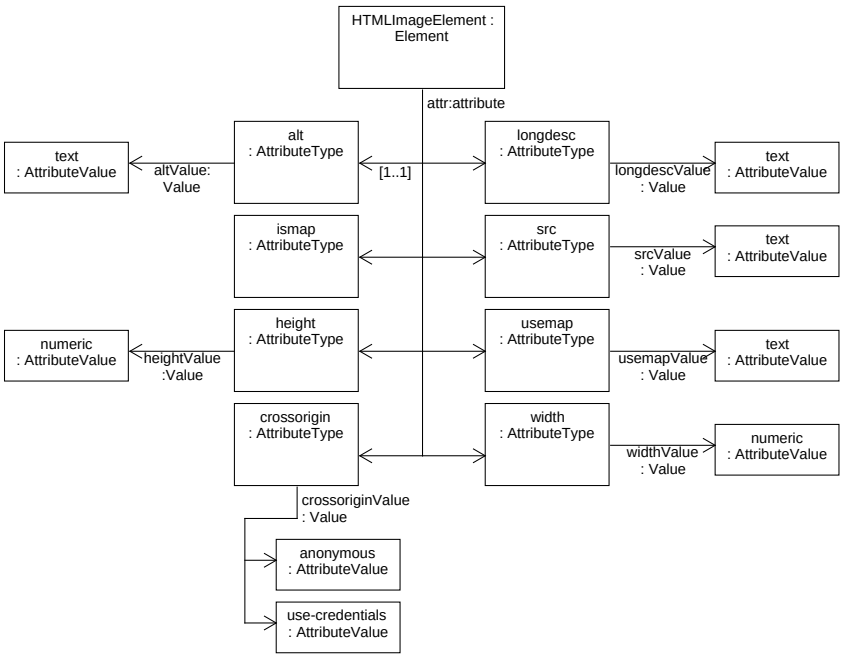


Figure 4.9: The Image Element Specification

The proposed solution to these three requirements is to include the *alt-attribute* in the html image element. The alt-attribute provides an alternative textual content which satisfies requirement 1.1.1 and 1.4.5 by providing an alternative text that can convey the information within the image, if properly formulated. Figure 4.9 shows that by using the *multiplicity predicate* and specifying that there must exist exactly one instance of the *alt*-node, it is possible to enforce the requirement of including the alt-attribute. Any instance that lacks the alt-attribute will not conform to this specification and be considered invalid.

Figure 4.10 shows an example of two image elements, the first one is invalid since it is missing an alternative text. The second one is valid since it does contain an alternative text describing the contents of the image.

Text and structure requirements

1.3.1 Info and Relationships [56] requires that the information, structure, and relationships conveyed through the presentation of the web page can be programmatically determined. The intention of this requirement is that the contents of the web page should remain constant despite how the page is viewed. For instance the common user would generally use a web browser to obtain the web pages contents visually, whilst a user with impaired vision would be required to use a screen reader to read out loud the contents of the web page. In these two scenarios it's important that the information the web page contains are interpreted equivalently by both users. In order to achieve this the browser or electronic aid might need to adapt the way the web page is presented in a way that satisfies the needs of the user.

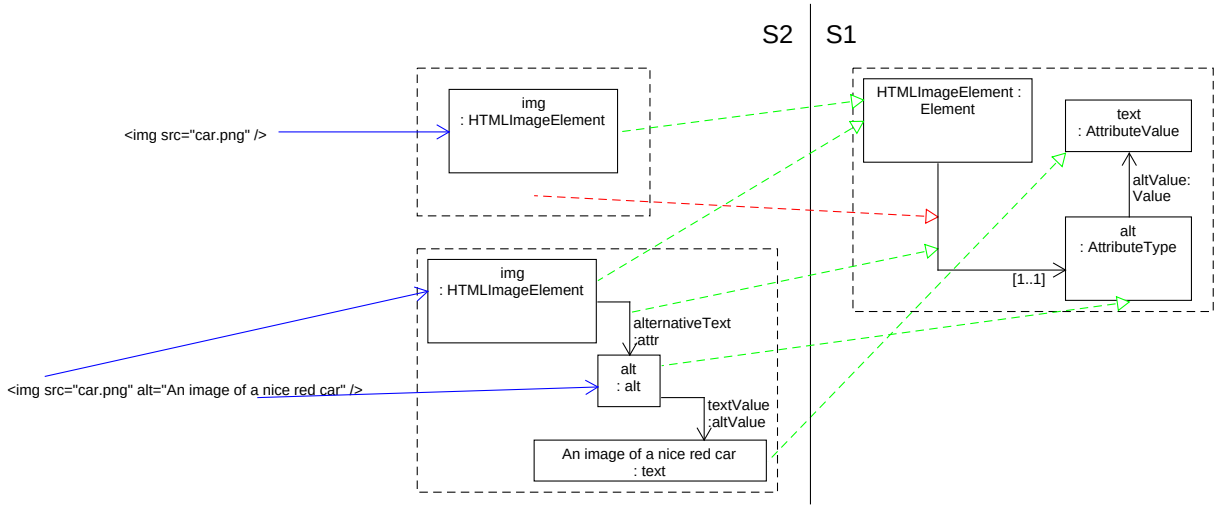


Figure 4.10: An example of two image elements, the bottom one contains the alt attribute and is therefore valid, the other does not and is flagged as invalid

2.4.6 Headings and Labels [56] states that headings and labels are used to describe topic or purpose. Descriptive headings can help a user find and understand the content of the web page. Labels can be used to provide descriptive information about components in the web page, such as input elements. Users with disabilities might interact with the web page using electronic aids such as screen readers, which often rely on the use of headers and labels to provide context to the page or it's components.

1.4.8 Visual Presentation [56] recommends that the user is able to modify the visual presentation of blocks of text using some sort of implemented mechanism provided by the web page, such as being able to maximise the contrast of the block of text by setting the background black, and font colour white.

2.4.10 Section Headings [56] recommends that section headings are used to structure the content of the web page. By using headers to break up the web page into smaller sections, the readability is improved and it becomes easier for the user to understand the content. Another important benefit is that by structuring the headers correctly, users who navigate the web page using a keyboard can more easily find the content they are searching for.

3.3.2 Labels or Instructions [56] states that a label or some sort of instruction needs to be provided for each input element the web page contains. As mentioned above, labels provide descriptive information about input elements that users with electronic aids are dependent of when interacting with the input element.

Some of these requirements are not possible to model, there is for instance no way to design a constraint that can confirm if a heading or a label provides an adequate description. It is however possible to ensure that headings, labels and other textual contents are implemented

correctly, thus ensuring that requirements 1.3.1 and 2.4.10 are met and that requirement 2.4.6 is partly met. In order to fulfil these requirements, DIFI suggests the following criteria [19]: *The order of the heading levels must descend sequentially and without skipping any levels.* Figure 4.11 shows an example of valid heading structure, and an invalid structure.

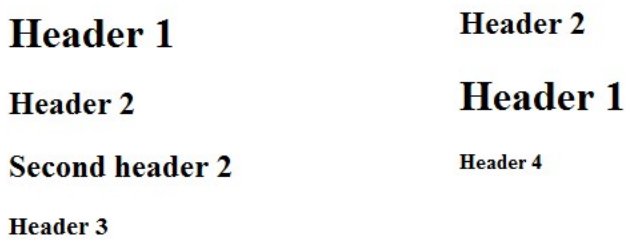


Figure 4.11: The headers to the left are correctly structure in accordance with the requirements specified by DIFI, the headers on the right violate those requirements.

In order to ensure that headings are correctly structured this thesis proposes adding a reference arrow from the header element h6 to h5, from h5 to h4 and so on until we reach h1. The header element h1 is the top level header and isn't dependent on the existence of another header element. The multiplicity constraint is added to each reference arrow to ensure that the source element is only valid if there exists a reference to the target element. Figure 4.12 displays how the heading specification may be modelled.

Furthermore, the label has a pretty clear purpose, it is suppose to provide additional information about the component it is attached to. If the label is not attached to any component, it should be considered as incorrectly implemented. A label is attached to another component by using the for-attribute that contains the ID of the element it is referencing, or by serving as a container for that element. Figure 4.5 shows how a label is correctly implemented by using the *Or predicate* [OR] to ensure that the label either contains the element it is attached to, or is referring to it using the for-attribute, or both.

It should be noted that the label element can only be attached to elements that are categorized as *labelable elements* [33]. With the exception of a few elements that are not included in our model, these are usually limited to input elements therefore our specification will only allow labels to be attached to input elements.

Finally in order to satisfy requirement 3.3.2 Labels or Instructions, the model ensures that each html input element has a descriptive text attached to it using either the *title*-attribute or by being referred to by a html label element. Figure 4.5 shows how the HtmlInputElement is correctly described.

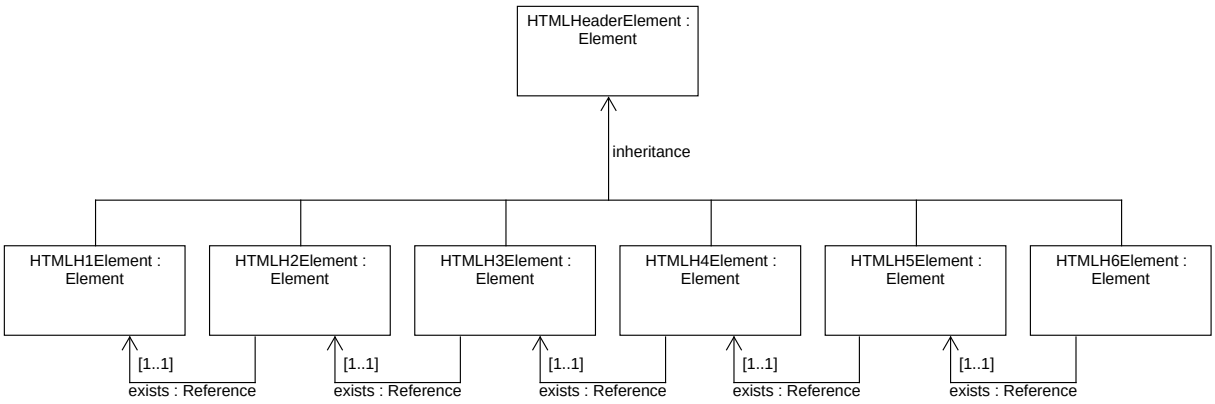


Figure 4.12: The Heading Element Specification

Contrast and styling requirements

1.4.3 Contrast (Minimum), in order to ensure adequate readability of the contents of the web page for users with impaired vision, there must exist a satisfactory contrast ratio between the various textual elements contained in the web page and their background. DIFI requires the following [17]: For large textual content such as headers, the ratio should be a minimum of **3.0:1**. For standard size textual content such as labels and body text, the contrast must be at least **4.5:1**. Figure 2.3 demonstrates the importance of having a sufficient contrast. The exception to this rule is for textual content that is used for decorative purposes only and not intended to provide any information to the user.

1.4.6 Contrast (Enhanced) is optional requirement that recommends the contrast ratio to **7:1** in order to achieve an adequate level of visibility.

In order to model this requirement a new specification for CSS rules is introduced, see figure

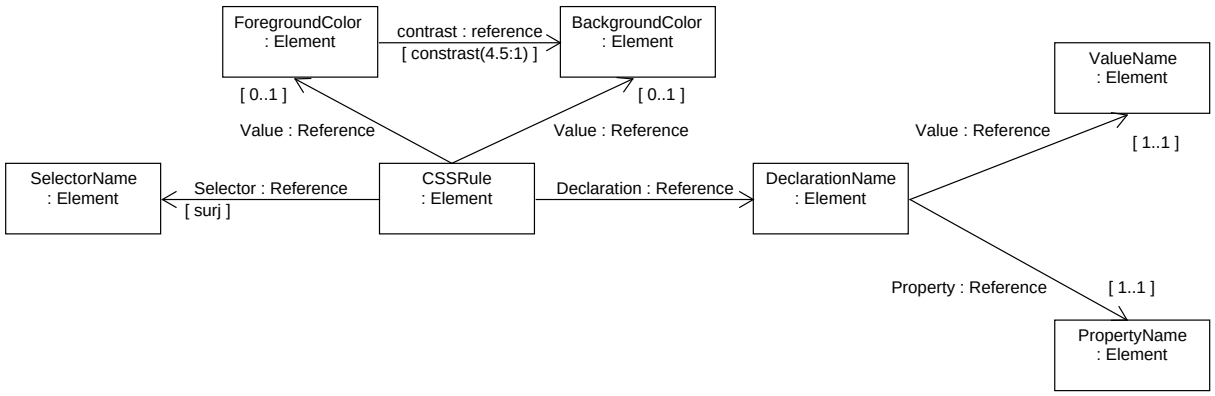


Figure 4.13: The CSS Rule specification

4.13. Before modelling the contrast requirement, modelling constraints must be added to the specification to ensure that the CSS rule is correctly formed. Figure 4.3 shows an example of the CSS syntax.

A valid CSS rule must contain a selector which is used to bind the style to a HTML element. The multiplicity predicate is used to enforce the presence of a selector. Furthermore, the CSS rule consists of a set of declarations which defines the style. Each declaration must consist of a property and a value, this is also enforced using the multiplicity predicate.

A reference arrow is added between the *ForegroundColor* element and the *BackgroundColor* element, with the *Contrast Predicate* $[contrast(m:1)]$ attached to it to ensure that the contrast ratio between two elements is sufficient. The value from the *color* declaration is mapped to the ForegroundColor element and the value from the *background-color* declaration is mapped to the BackgroundColor element. Using the values from these two declarations we can calculate the ratio using the following formula [58]:

$$ContrastRatio = \frac{L1}{L2} = \frac{(0.2126 * R_1) + (0.7152 * G_1) + (0.0722 * B_1) + 0.05}{(0.2126 * R_2) + (0.7152 * G_2) + (0.0722 * B_2) + 0.05}$$

where R,G and B are defined as

if $R_{sRGB} <= 0.03928$ then $R = R_{sRGB}/12.92$ else $R = \frac{(R_{sRGB} + 0.055)^{2.4}}{1.055}$

if $G_{sRGB} <= 0.03928$ then $G = G_{sRGB}/12.92$ else $G = \frac{(G_{sRGB} + 0.055)^{2.4}}{1.055}$

if $B_{sRGB} <= 0.03928$ then $B = B_{sRGB}/12.92$ else $B = \frac{(B_{sRGB} + 0.055)^{2.4}}{1.055}$

and R_{sRGB}, G_{sRGB} and B_{sRGB} are defined as

$R_{sRGB} = R_{8bit}/255$
$G_{sRGB} = G_{8bit}/255$
$B_{sRGB} = B_{8bit}/255$

m is the lowest acceptable contrast ratio, RGB_1 is the based on the value of background-color declaration belong to the target element, and RGB_2 is based on value of the color declaration to the source element. The $ContrastRatio$ value is somewhere in the range of 1.0 where there is no contrast at all (i.e. black on black, or white on white), and 21.0 where there is maximum contrast (i.e. white on black). If $ConstrastRatio < m$, the constraint is violated, the html element should be considered invalid.

For instance a html element with the following color value $\#FFFDAA$ validated against a

html element with the background-color value $\#FFFFFF$ will give the following contrast ratio:

$$CR = \frac{(0.2126 * 1.0) + (0.7152 * 1.0) + (0.0722 * 1.0) + 0.05}{(0.2126 * 1.0) + (0.7152 * 0.9822) + (0.0722 * 0.4019) + 0.05} = \frac{1.05}{0.994} = 1.0563 \approx 1.1$$

1.1 is below the acceptable ratio of 4.5 and therefore violates the contrast constraint. However, if the color value had been darker such as #444444 and the background-color remained $\#FFFFFF$, the contrast ratio would be:

$$CR = \frac{(0.2126 * 1.0) + (0.7152 * 1.0) + (0.0722 * 1.0) + 0.05}{(0.2126 * 0.0578) + (0.7152 * 0.0578) + (0.0722 * 0.0578) + 0.05} = \frac{1.05}{0,1079} = 9.7483 \approx 9.7$$

9.7 is a larger than the minimum allowed ratio 4.5 and therefore the html element satisfied the constraint.

4.4 Summary

Table 4.1 displays an overview of how many of the requirements were able to be included in the model. The table shows that five out of the nine requirements selected for this thesis were included in the model. The two most critical requirements included in the table with a criteria *level A* were successfully implemented into the model. Furthermore, of the five requirements that are mandatory, three of them are included in the model and one of them was partly included.

Requirement	**Included in model**	**Mandatory**	**Criteria level**
1.1.1 Non-text Content	Yes	Yes	A
1.4.5 Images of Text	No	Yes	AA
1.4.9 Images of Text (No Exception)	No	No	AAA
1.3.1 Info and Relationships	Yes	Yes	A
2.4.6 Headings and Labels	Partly	Yes	AA
1.4.8 Visual Presentation	No	No	AAA
2.4.10 Section Headings	Yes	No	AAA
3.3.2 Labels or Instructions	Yes	Yes	A
1.4.3 Contrast (Minimum)	Yes	Yes	AA
1.4.6 Contrast (Enhanced)	Yes	No	AAA

Table 4.1: Table of selected requirements

Despite not being able to model all of the selected requirements, over half of them were included in the model. In addition the model ensures that the web page is syntactically correct, by checking for instance that the web page only contains valid html elements and attributes, that only certain html elements are allowed to contain other elements, and that headers and labels are correctly implemented.

When defining the model using DPF in the previous section, several challenges were encountered: The first challenge was ensuring that the requirements were interpreted correctly and thus reflected correctly in the model. All of these requirements obtained from the WCAG 2.0 standard, are written in a natural language and are therefore open to misinterpretation. However, many of these requirements contain a detailed explanation and how the requirement is evaluated. The additional information about the requirement combined with possible corresponding solutions suggested by DIFI made it possible to design a fairly accurate model, but there is still a danger that the formal definition provided by the model might be inconsistent with what a human examiner's interpretation of the requirement.

The second challenge was modelling requirements that focus on the actual contents of the Html Element such as 1.4.5 Images of Text, where due to technological limitations it isn't possible to determine if an image contains any textual content, or 2.4.6 Headings and Labels where it is difficult to determine if headings and labels provide an adequate description of the content they are describing. This could be achieved using advance image processing techniques or with the help of an AI. However, validating these sorts of requirements would have to be done outside the scope of the model and the DPF framework.

The primary task of the WCAG 2.0 model described in this chapter is that it can provide a clear and precise definition of the requirements included in the evaluation process. The WCAG 2.0 model uses a graph-based graphical syntax to represent the requirements included in the evaluation process. The advantage of using a graphical syntax is that it is easier for domain experts to interpret its contents, thus the WCAG 2.0 model can be used as a communication medium between the domain expert and the developer. However, it is possible that some explanation might be needed initially when interpreting the graph-based model, but ultimately describing the requirements as a graph consisting of nodes and edges is a great deal easier to understand than for instance a XSD and a lot more precise than a definition written in a natural language.

Chapter 5

Model Driven Web Page Validation

The previous chapter described a meta-model that provides a formal definition of the requirements a web page must fulfil. This chapter will discuss two approaches that will use the meta-model from chapter 4 to validate an existing web page. These approaches are referred to as the *Top-Down Model Validation Approach* and the *Bottom-up Model Validation Approach*. This chapter will conclude with a short discussion of which approach is best suited for evaluating an existing web page when considering the challenges presented in chapter 2.

Before describing the two approaches, it is important to clarify the two separate spaces the solution spans across. The first space consists of technology dependent artefacts such as source code, web pages and other platform specific artefacts, this space is referred to as the *technical space*. The second space contains only technology independent components such as graph-based models, and is referred to as the *conceptual space*. Considering the two spaces and their contents, it can be assumed that a domain expert without the necessary technical knowledge, will only be able to understand the contents in the conceptual space, but not the components in the technical space. See figure 5.1.

5.1 Top-Down Model Validation Approach

The Top-Down Model Validation Approach, hereby referred to as the *Top-Down Approach*, attempts to generate the functions based on the WCAG 2.0 model, that the evaluation process is comprised of. The idea is that by using a meta-model, it is possible to generate some sort of logic for the web evaluation tool based on the constraints and structure of the meta-model. This could be for instance functions that will parse the provided HTML document to determine if the web page is valid or not.

This approach builds upon the specifications defined by MDA, described in section 3.1.2. The WCAG 2.0 model functions as the PIM, and the generated source code would be classified as the PSM. This means that basically, the Top-Down approach consists of a *forward engineer-*

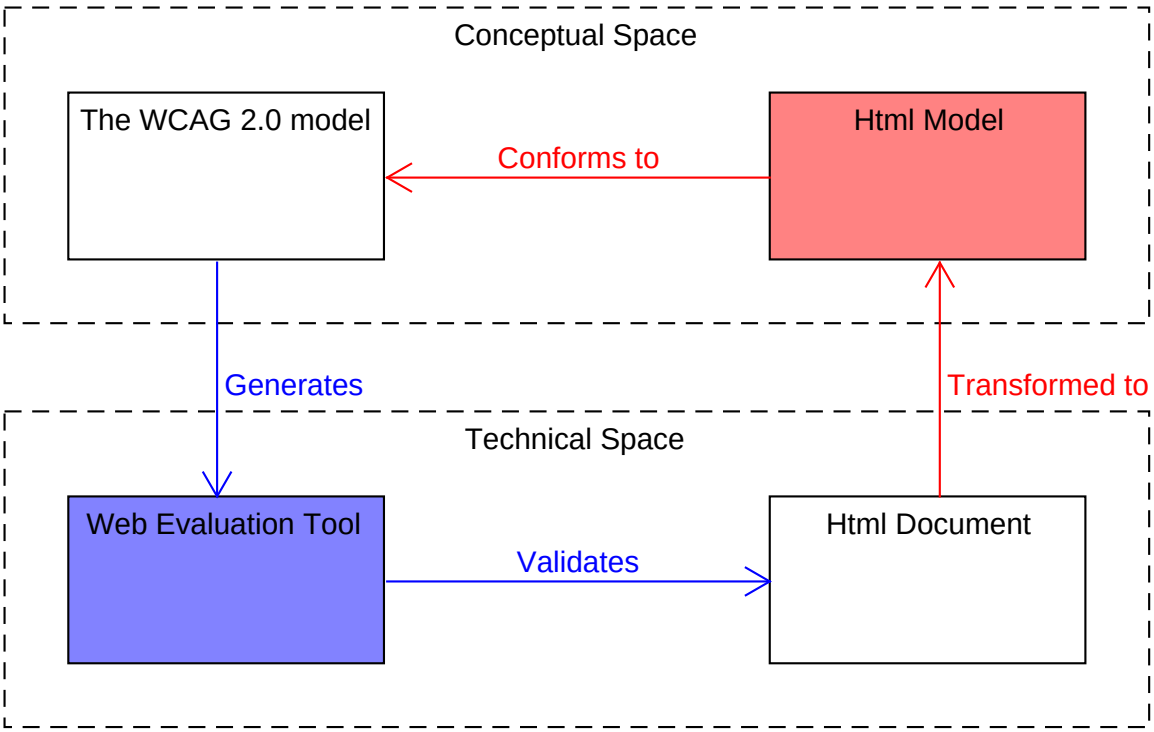


Figure 5.1: Conceptual view of the two spaces the solution spans across

ing process that uses an abstract, platform-independent model, to generate a platform-specific piece of source code. The CIM in this scenario would be the requirements defined in a natural language.

In order to understand the Top-Down Approach, this section will first describe the fundamentals of *Model-Driven Code Generation*, before providing a detailed explanation of the actual approach.

5.1.1 Model-Driven Code Generation

As briefly mentioned in chapter 3, code generation is defined as a type of Model-To-Text transformation which aims at generating executable code from a higher level model. Code generation within MDE is a concept adopted from programming compilers that are primarily tasked with transforming high-level programming languages to machine code. Code generation differs from the compiler by transforming models into source code, thus the code generator (sometimes referred to as the *Model Compiler*) can be stated as being built on top of existing compilers for programming languages. In other words code generation can be defined as a vertical transition from a model at a higher-level of abstraction to a lower-level artefact such as source code. Basically a code generator is a program designed to write another program [10].

Advantages and Drawbacks

There are several advantages with model-driven code generation [10]:

- *Protects the modellers intellectual property.* Code generation allows the developer to deploy a running application for a specific client without sharing the conceptualization and design. This isn't really an advantage relevant for the challenges presented in chapter 2.
- *Can generate code to Multiple platforms.* Code generation isn't locked to one vendor or platform, the developer can easily change the runtime environment without making changes to the model. This means that the proposed HTML-validator can be generated to the Java platform, the ASP.NET platform or even a pure JavaScript project using the same model.
- *Can reuse existing programming artefacts.* Code generation can generalize existing pieces of code and use it as a template when generating new parts of the system.

Perhaps one of the greatest advantages with code generation is the ability to produce repetitive code automatically with minimum errors. This allows the developer to ignore certain bad habits such as code duplication. Furthermore, the fact that the code is generated by a code generator rather than manually written may reduce the number of errors or bugs in the source code, contributing to an overall improvement to the quality of the software [28].

There are however, despite the numerous advantages with the code-generation approach several drawbacks. The biggest one is that the generated code is often structured and / or written in a way that may look unfamiliar to a developer. This can vary from small things such as a different line indentations, to larger things such as a function written in a way that may seem illogical. Unless the developer spends a lot of time adjusting the output of the code generation, the result will most likely not fit in with the general style of the system [10].

The Code Generation process consists of two environments: The first is *The Target Environment* where the generated code is inserted and it's context which can be an empty new project, or an existing project which the generated code is to be implemented into. The second is *The DSL processor* which consists of the semantic model, the parser and the code generator [28].

The main point of using code generation is to separate the target environment from the DSL processor, this is because of potential limitations in the target environment that might prevent an adequate DSL processor from being developed. The target environment might for instance, lack the resources to run the DSL processor or defined in a language not suitable for the DSL processing. Another reason may be the lack of familiarity with the target environment. It is generally easier to specify behaviour in a more familiar language and then duplicate that behaviour in the less familiar language using code generation [28].

There are three essential questions that should be addressed when developing a model-based code generator [10]: *How much is generated?*, *What is generated?* and *How it's generate?*

5.1.2 What to Generate

When choosing what to generate, it should be noted that generally the less code generated to represent the system the better, this is because generated code is more difficult to edit than handwritten code. This can be achieved by taking advantage of available frameworks and APIs. It is also considered good practice to have a clear idea of what is to be generated beforehand in order keep the generated code as concise as possible [10] .

It is possible to differentiate between two general styles of code generation when determining what kind of code to generate: *Model-aware Generation* and *Model Ignorant Generation* [28].

Model-Aware Generation

The Model-Aware Generation style is used when the meta-model is an abstract representation of an artefact in the target environment, the goal here is to preserve the principle of generic-specific separation. In other words, the generator is able to keep the generic framework code and the specific configuration code separate. This is one of the advantages with the Model-Aware Generation, and is done by only generating the specific configuration code based on the semantic model into the target environment, and that the generic framework code is implemented manually by the developer (the generic framework can also be generated, but is done separately). Figure 5.2 illustrates how the Model-Aware Generation style works. Generally if the code generator is generating code to work with a model, then it's using Model-Aware Generation [28].

Considering the domain presented in chapter 4, an example of the Model-Aware Generation would be generating a C# class representing an html element and inserting it into an existing web application project. Instances of the class would then represent certain html elements in a web page.

Of the two styles presented in this section, this style is considered to be the preferred one. By having a representation of the model both in the the DSL processor and the Target Environment, code generation becomes easier to build, test and maintain. The only times not to use the Model-Aware generation style is when the target environment is limited and therefore difficult to create a representation of the semantic model in the target environment at runtime [28].

Model Ignorant Generation

Not all Target environments can facilitate a representation of the meta-model. This may be due to limitations in the target language or in the environments resources. The Model Ignorant Generation Style is suited for situations where it is not possible to use the Model-Aware Generation style. The Model Ignorant Generation generates code without interacting with an instance of the meta-model in the target environment. Unlike the Modal-Aware Generation style, the

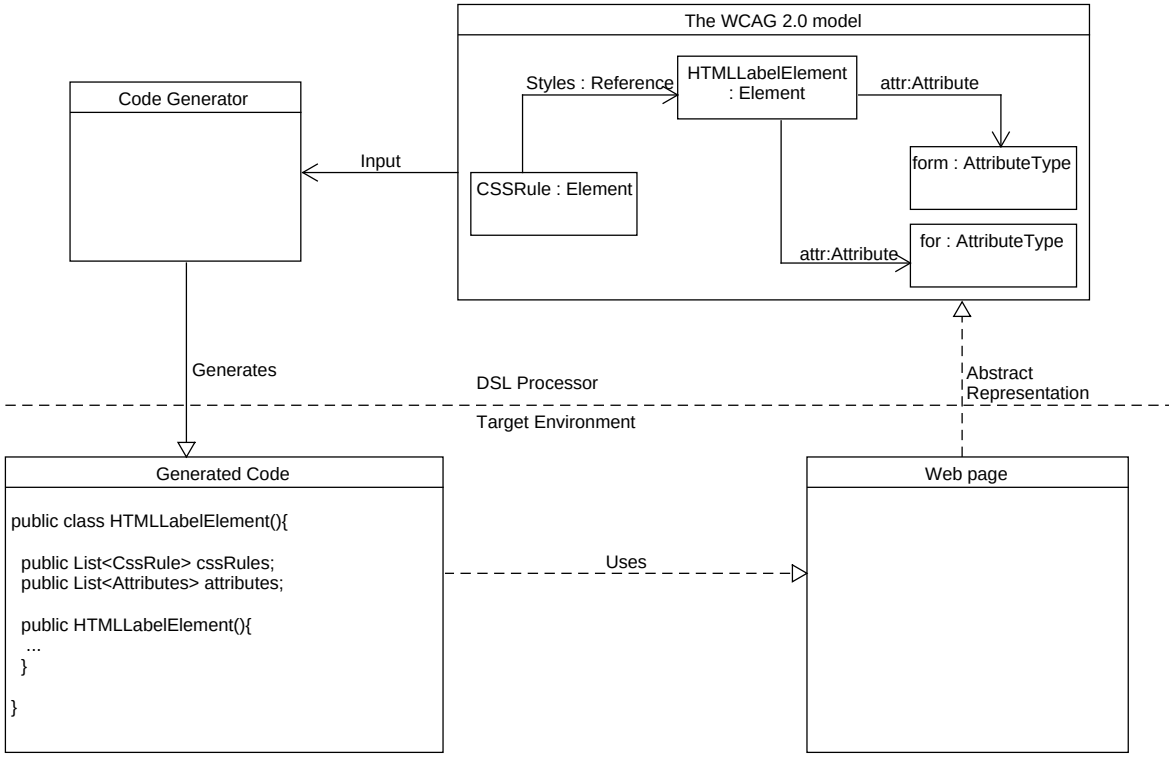


Figure 5.2: An example of the Model-Aware Generation Style

Model Ignorant Generation style does not make any distinction between the generic framework code and the specific configuration code. This means the Model Ignorant Generation style often ends up generating more code than the Model-Aware Generation style. The advantage with this style is that the developer does not have to spend time making complex data structures, instead focusing on procedural code and simple structures [28].

Generally the Model-Aware Generation style is easier to use when generating code, since it normally results in a generation process that is simpler to understand and modify. But since the Model Ignorant Generated code is procedural, it's easier to follow of the two styles. Generally we use Model Ignorant Generation in situations where Model-Aware Generation isn't possible.

Figure 5.3 shows how the Model Ignorant code generator creates code that does not interact with any artefacts represented in the DSL Processor.

5.1.3 How to Generate

Once it has been determined what parts of the code should be generated and in which target language, the developer has to decide how generate the source code. Again, it is possible to differentiate between two main styles when generating textual output: *Transformer Generation* and *Templated Generation*. However, these styles are not necessarily exclusive of each other,

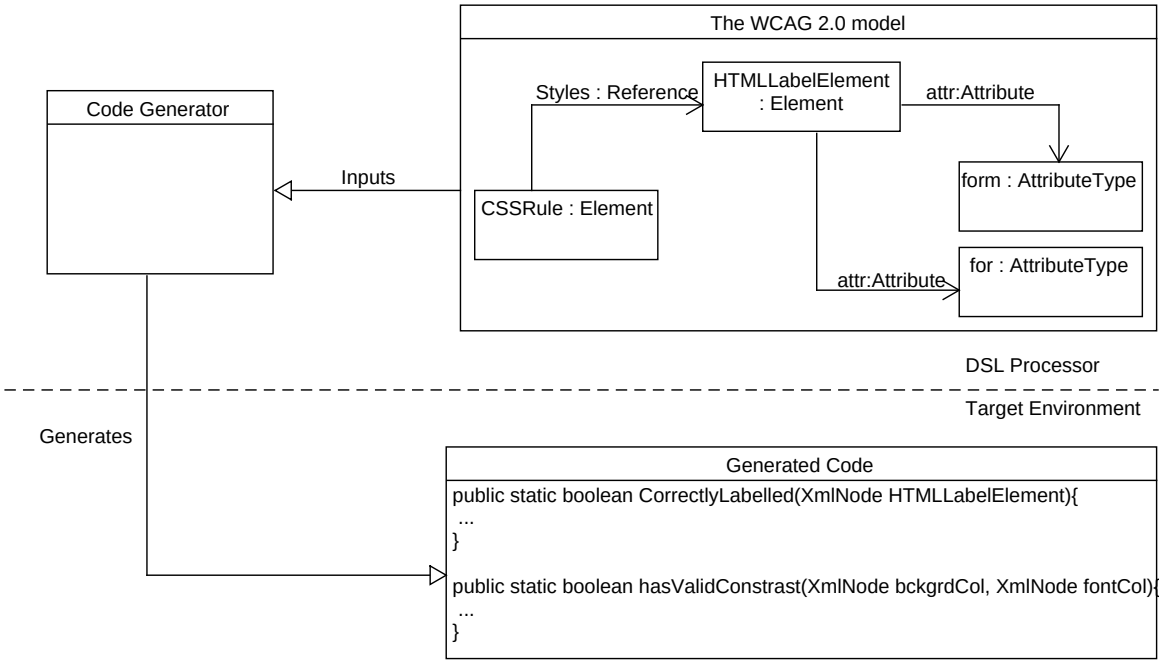


Figure 5.3: Example of the Model Ignorant Code Generation Style

and more often than not, code generation processes consist of a combination of the two using an *Embedment Helper* [28].

Transformer Generation

The Transformer Generation Style involves developing a program that uses a Meta-Model, such as the WCAG 2.0 model, as an input, and outputs a piece of generated source code in the target environment. This is the recommended style for situations where the output text has a simple relationship with the input model and most of the output text is generated. For more complex relationships it's possible to run the transformer generation in multiple stages, where each stage handles a different aspect of the problem [28].

When generating functions to be included in the evaluation process, a possible approach is to generate a function for each modelling constraint included in the WCAG 2.0 model which

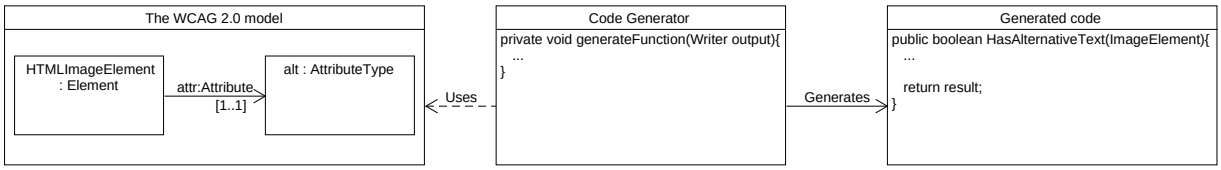


Figure 5.4: Example of the Transformer Generation Style

inputs the html elements that correspond to the source node and the target node. For instance, a generated function that wishes to verify that every image element has an alternative text could use an html image as input at determine that it contains exactly one alternative text, see figure 4.9. An example of the Transformer Generation style is illustrated in figure 5.4.

Code generation processes that adopt the Transformer Generation style, can be divided up into the following phases when generating code with a DSL parser implemented in a object-oriented programming language [10]:

- *The Load Phase.* In the first phase the model is deserialized from it's XMI representation to an object that can be manipulated using the programming language.
- *The Code Production Phase.* Using the deserialized model from the previous phase, the necessary information is collected for generating the code.
- *The Code Writing Phase.* In the final phase the information gathered in the previous phase is used to output the code to a file.

Figure 5.5 illustrates an example of the code generation process. In the load phase, the model is parsed by an XMI parser (if working with an external DSML), or accessed with the models API (if the DSML is internal). Then by using code generation techniques, a textual piece of code is generated. The process is concluded by the Java *FileStream* component which outputs the generated code to a file.

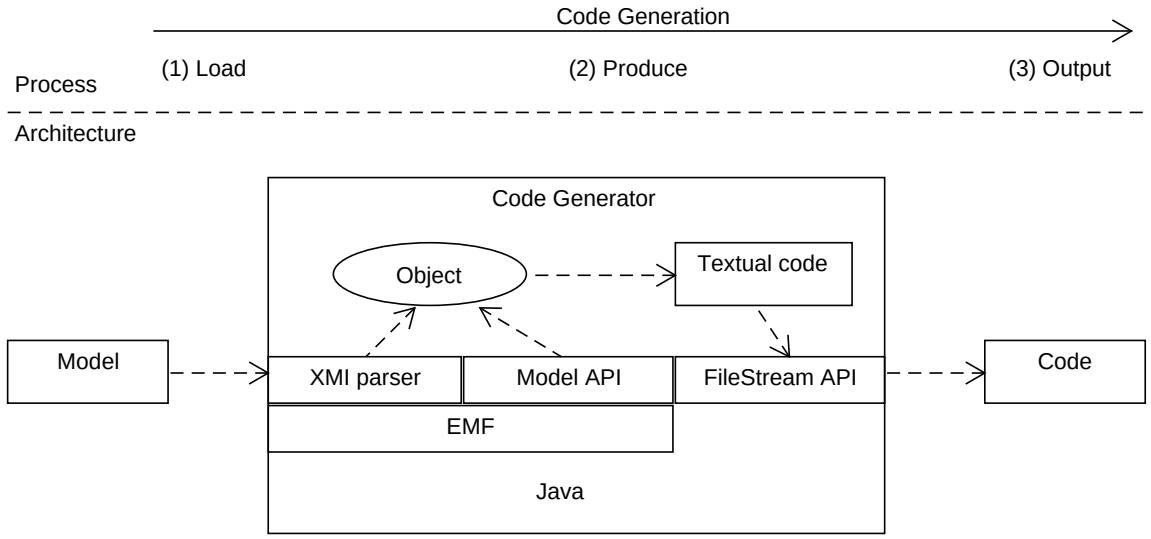


Figure 5.5: Example of code a generation process [10]

Templated Generation

The Templated Generation Style involves using an existing sample output file containing special template markers that references specific parts of a meta-model to generate the appropriate code. By defining a sample output file the code generator has a pretty clear picture of how the result will look like, because of this the Templated Generation style can be defined as structure driven (as opposed to the Transformer Generation style that is input or output driven) [28].

The Templated Generation Style is actually frequently used in web development frameworks such as ASP.NET Webforms and Java JSP. In these frameworks the .aspx and .jsp files function as the template which include references to code on the server-side, at runtime the result of these references is rendered to the client as part of the HTML page. [28].

The Templated Generation style can be broken down into three main components: *the Template*, *the Context* and *the Templating Engine*. The template is the sample output file which contains the special template markers which are replaced with data from an instance of the meta-model during the code generation process by using the context [28].

The context is normally a simple data structure that contains the data the template requires, however in some cases it might be necessary to retrieve data based on some sort of conditional or to output a list of items, for this it is possible to use the *Embedment Helper* described in section 5.1.3 [28].

Finally the Templating Engine is the component that combines the template and the context by replacing the special template markers with the data contained in the context, thus creating a complete output file when executing the code generator [28].

The advantage of dividing the style into three separate components is that it is possible to run the same template with several different contexts in order to generate multiple output files. Another advantage is that by using the Templated Generation Style it is possible to create a separation between the static code (code that doesn't depend on the context), and the dynamic code (code that changes based on the context). This is one of the drawbacks with the Transformer Generation Style [10].

The Templated Generation style is considered best suited when there is a lot of static code mixed into the output. This is also the preferred style when using the Model Ignorant Generation Style, since the DSL parser is basically just filling in the blank spaces in the sample output file.

Embedment Helper

The main goal of the Embedment Helper is to minimize the code in a template by providing all the needed functions the template needs. As explained in the previous section there are scenarios where the output is determined by a conditional or that the output may need to use an iterator to output a list of items. Generally it is considered good practice to keep the template

as tidy as possible so that the host code is clear, and in order to do that all functions required by the template are grouped in an Embedment Helper [28].

5.1.4 How Much to Generate

When determining how much to generate, there are two central terms that describe the extent of the code generation process: *Full code-generation* and *Partial code-generation* [10]. A full code-generation means that it is possible to automatically generate the entire source code. This implies that all changes to the meta-model are reflected in the source code, something that can be very beneficial in many cases. Unfortunately it is seldom that the meta-model contains enough information to achieve a full code-generation.

A partial code-generation means that only parts of the source code is automatically generated, meaning that the developer has to complete the source code by filling in the blanks left by the code generator manually. This can lead to problems if not properly handled, since ideally the source code should reflect the meta-model as closely as possible.

In order to avoid problems involving manually written code mixed with generated code, two general rules are suggested: *Don't modify generated code* and *Keep generated code clearly separated from manually written code* [28].

One way to handle mixed manually written code and generated code is by defining protected areas that contain only manually written code [10]. An alternative is to go one step further and keep the generated code and manually written code in separate files by using *the Generation Gap technique* [28].

Generation Gap

Generated code should never be manually edited by hand, otherwise manual changes to the code will be lost during the next code generation process. The main purpose with the Generation Gap is to keep the generated and manually written parts in separate classes and linking them using inheritance, thus it is possible to keep the two separate from each other. The general idea is to generate the superclass, then manually code the subclass. This way the developer can easily call any generated feature needed and override other features if necessary [28].

Obviously this is a style that can only be used in target languages that supports inheritance. However, it is a very effective technique for splitting one logical class into multiple separate files in situations where parts of the logic is manually written and other parts are generated [28].

5.1.5 The Top-Down Model Validation Process

As mentioned in the introduction to this section the *Top-Down Model Validation* approach attempts to generate functions that comprise the evaluation process, based on the information provided by the meta-model. In this case the meta-model is the WCAG 2.0 model defined in chapter 4. Using the WCAG 2.0 model, it should be possible to generate the logic for the evaluation process that can be inserted into the web evaluation tool. This means that the code generation process can be classified as a code first approach. One thing that should be noted about this approach is how it attempts to use information available in the conceptual space in order to solve the problem in the platform specific technical space. The transition from the conceptual space to the technical space is achieved using Model-Drive Code Generation.

What is generated?

Considering the code generation process described earlier in this section, the first aspect that should be addressed is *What is generated*? The question here is really if the code generation process is Model-aware or Model ignorant, before this can be answered it should be determined what exactly is being generated. The generated evaluation logic needs to be able to do two things:

- Interpret an html document or a representation of it
- Determine if the html document is valid.

The proposed solution is to generate functions based on the modelling constraints in the WCAG 2.0 model, that can determine if specific elements in a web page meet the necessary requirements or not. Here there is two obvious choices: The code generator can generate either platform specific functions such as methods in C#, or generate functions in an external language such as XQuery FLOWR functions [21].

Regardless of if the generated function is platform specific or not, it is safe to assume that the code generation process is Model-aware. This is because the purpose of the generated code is to interact with an instance of the WCAG 2.0 model, which happens to be an abstract representation of an arbitrary web page that exists in the target space.

How much is generated?

The second aspect that needs to be addressed is *How much is generated*? In the summary of chapter 4 it was noted that several of the requirements were not possible to include in the model (see table 4.1), it is therefore only reasonable to assume that parts of the evaluation process is manually written by developers to include functions that can confirm for instance if the alternative text adequately describes the image. Furthermore, the evaluation process will require some

sort of html interpreter that fetches an arbitrary web page and presents it in a form that can be analysed by the evaluation process. In the context of this approach, it makes sense that only the code that is represented in or affected by, the WCAG 2.0 model is generated, while the rest is manually written.

Since it is not possible to include off all of the selected requirements in the WCAG 2.0 model, and therefore some of the mandatory requirements must be implemented manually, it is possible to determine that the Top-Down approach consists of a partial code generation process.

How it's generated?

The final aspect to be considered is *How it's generated*? Which is better suited; Templated Generation, Transformer Generation or a combination of the two? Generally the transformer generation is better for simpler code generations where there is a clear relation between the source code and the model.

Had the Top-Down Model Validation approach been limited to only generating an alternative representation of the WCAG 2.0 model in the applications host language, the Transformer Generation would of been sufficient. However, it is desirable that the generated code contains validation logic that can determine if the web page is valid or not, making the generation rather complex.

A pure Templated Generation approach might also be problematic, since again there is a great deal of logic that needs to be included in the generation. Without using Embedment Helpers, the template will most likely become polluted with foreign template specific code, making the template difficult to read and maintain.

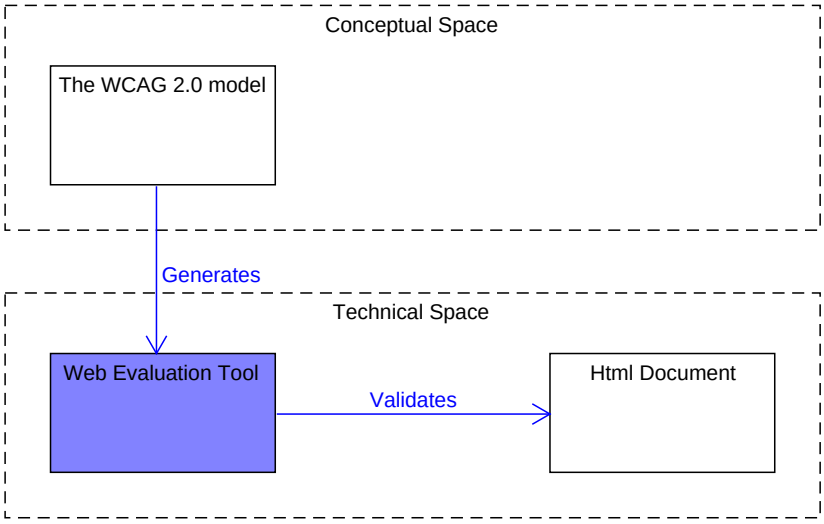


Figure 5.6: Illustration of the top down validation approach

The optimal solution is a combination of the two styles. The Templated Generation handles the structure and the static elements of the document based on the contents of the WCAG 2.0 model, the transformer generation is implemented using embedment helpers that can handle more specific code generation determined by logic for instance conditionals.

The Process

The Top-Down Model Validation Process can be broken down into two steps: (1) Generating the validation logic, and (2) Validating the Html document.

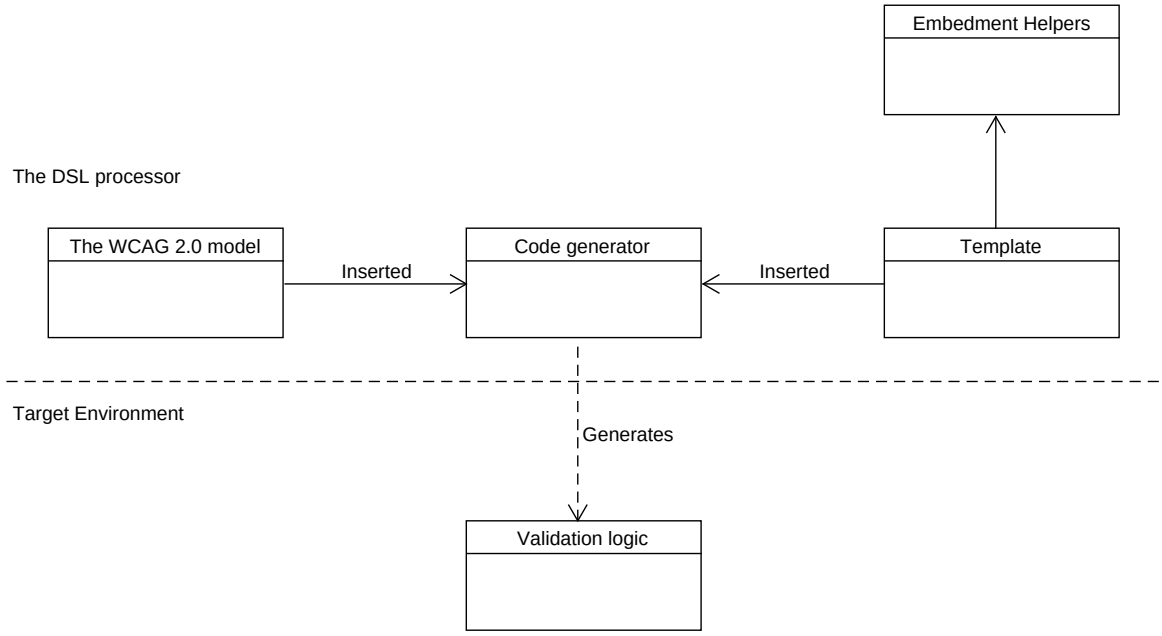


Figure 5.7: Step one: The code generator uses the WCAG 2.0 model and the Template to generate the validation logic

In *step one*, the WCAG 2.0 model and a manually written template with embedment helpers is inserted into code generator. The combination of these two provide the code generator with the necessary information to generate validation logic that can be inserted into the web evaluation tool. See figure 5.7.

The mandatory requirement *1.1.1 Non-text Content* requires that all non-textual content such as an image is described by some sort of textual content. The suggested solution is to describe a html element such as an image, using the alt-attribute. This requirement is formalized in the WCAG 2.0 model using the multiplicity predicate, see figure 4.9. During step one, the code generator will parse the model, and generate an XQuery function that will loop through all image elements and return any image element that does not contain exactly one alt-attribute.

In *step two* the validator uses the generated validation logic to confirm if the web page is

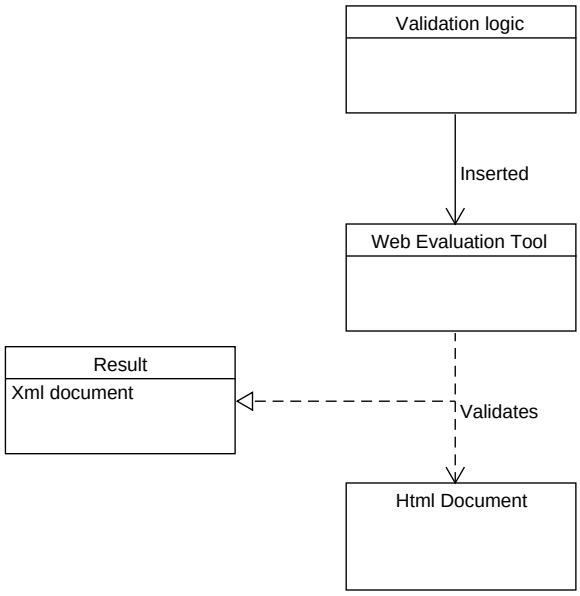


Figure 5.8: Step two: The validation logic is inserted into the validator. The validator uses the generated logic to determine if a web page is valid or not

valid or not. All identified errors are stored in an Xml document which can be used to inform the user of errors found within the web page. See figure 5.8.

Continuing on the previous example, during step two, the XQuery function that finds and returns all image elements not containing exactly one alt-attribute is executed upon the web page. Any element that is returned is considered invalid and stored in a common XML document.

5.2 Bottom-up Model Validation Approach

The basic idea behind the Bottom-up Model Validation approach, hereby referred to as the *Bottom-up approach*, is to extract an abstract model representation of the web page using *Reverse Engineering (RE)* combined with MDE. The extracted model can be manipulated, transformed and validated using techniques from MDE. By confirming that the model is valid, the web evaluation tool can claim that the corresponding web page is also valid.

The term Reverse Engineering originally referred to the process of deciphering designs of an existing hardware product in order to improve another product, or to analyse a competitor's product. This process would be carried out by someone else than the original developer and without out the aid of the products schematics. Within the software engineering domain, reverse engineering is a concept usually associated with the process of examining and modernizing a legacy system. Where the goal of reverse engineering a hardware component is to reproduce it, reverse engineering a software component provides a better understanding of system in

order to maintain it, improve it or refactor parts of it [15].

Reverse engineering can be defined as "The process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [15]. In other words it can be described as the process of examining an existing artefact and extracting it to a more abstract representation of itself in order to complete a task. Note that the actual process of reverse engineering examines the artefact and does not modify or replicate it in any way.

When attempting to reverse engineer a web page, it is important to consider the degree of dynamic behaviour that should be included in the abstract model. In other words, should the abstract model contain events and functions that alter the web pages structure and style, or should it rather be an abstract model of a snapshot of the web page. It is possible to shed some light on this problem by dividing web pages into three separate classes [53]:

- *Class 1*: Static HTML Web pages consisting primarily of HTML code. A Class 1 web application is the easiest class category to reverse engineer because of their limited functionality and lack of user interaction.
- *Class 2*: Dynamic HTML Web pages with client side functionality provided usually by JavaScript. Class 2 web applications are slightly more difficult to reverse engineer because of dynamic behaviour caused by JavaScript.
- *Class 3*: Web applications with Server side functionality. These applications are considered the most complexed of the three classes. Typically a JAVA EE or ASP.NET application would belong to this category.

The web page domain is defined in chapter 4 as a combination of three sub-domains: The HTML sub-domain, the CSS sub-domain and the Script sub-domain (see figure 4.1). Furthermore, the selected requirements focus primarily on the static parts of the web page, which means during the evaluation process, it is only the structure (defined by the HTML document) and style (defined by the CSS stylesheet) of a web page that is of interest. Any user interaction with the web application such as submitting the form is considered irrelevant in this context, therefore this approach will treat the web page as a *Class 1 web page*. This also implies that the WCAG 2.0 model is categorized as a static model, rather than a dynamic model.

The following sub-sections will explain the core principles and process of Model Driven Reverse Engineering and what challenges must considered when using MDRE to reverse engineer an artefact such as a web page. This section will conclude with a description of the Bottom-up Model Validation process.

5.2.1 Model Driven Reverse Engineering

Within the discipline of meta-modelling the abstract view of a web page is represented as a model. The term *Model Driven Reverse Engineering (MDRE)* was initially introduced as a method to predict the amount of time it would cost to reverse engineering a legacy system and measure the quality of the result [47]. MDRE applies core techniques and principles from MDE to the abstract model of a legacy system. This provides the means to move the problem from the technical space to the conceptual space. This means it will be possible to evaluate the web page regardless of whatever technology it's implemented with (e.g. JSP, ASP.NET, HTML, XML etc.). Moreover, by representing an artefact as a model, it provides a better understanding of the artefact which simplifies the process of examining and if necessary modifying the artefact.

The main role of the MDRE process is extracting information from an existing system to obtain a better comprehension of that system. The source code to a system can be described as the most accurate and reliable description of how that system behaves [54]. Even though diagrams and documentation are produced during the initial development phase, these should be considered outdated as soon as the software is modified. The result of the Reverse Engineering process is an abstract view of the system to aid the developer in maintaining that system. The result is normally a diagram intended for a human to interpret and not a computer. This is where the scope of the Reverse Engineering process ends. In other words Reverse Engineering can be characterized as a process of generating documentation that aids a developer in comprehending the software system.

The result of the MDRE process is a model representation of the system, the model can still be used to fulfil the traditional task of providing the developer with documentation for how the system works, but may also be used in a variety of different MDE applications such as code generation and enabling system interoperability [10].

MDRE is commonly defined as the application of MDE principles and techniques to the reverse engineering process. The main difference between these two methodologies is that MDE focuses on Forward Engineering where models are used to represent a system at a higher level abstraction and using the models to implement the system, MDRE suggests doing the opposite and create the high level abstract models based on an existing system. This is illustrated in figure 5.9 [53].

Even though the domain and the challenges described in the chapter 2 does not involve a legacy system, many of its principles and methods are relevant when designing the solution. Since the end goal of our MDRE process is to evaluate an existing system and not reengineer an obsolete legacy system the target system will be referred to as the artefact instead of the legacy system.

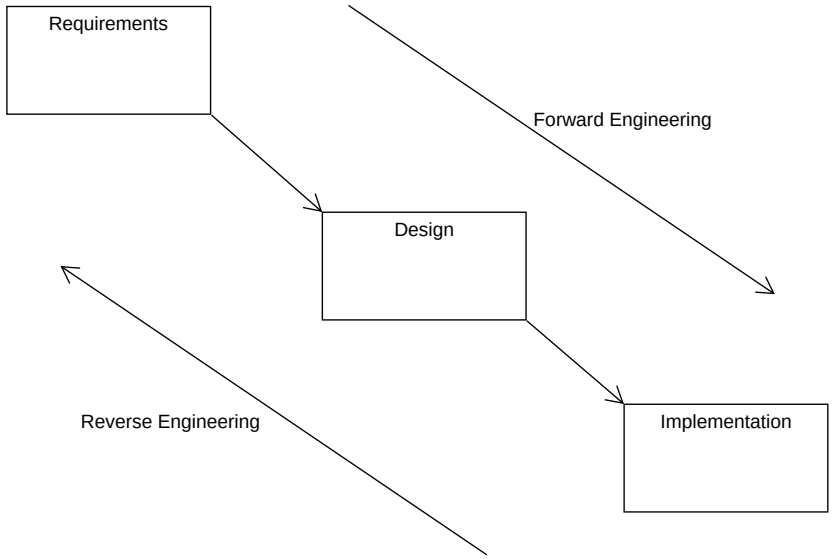


Figure 5.9: An example of a Reverse Engineering process

Challenges

Reverse engineering an artefact such as a web page can be a complex process, and in order to ensure that the extracted abstraction is sufficient and correct, there are several challenges that must be addressed [12]:

- *Information loss* is a common problem that involves retrieving information from a system. Most software systems are built up of a number of diverse components, ensuring that the quality and that all the necessary information has been extracted from the system can be a rather complex task. A web application for submitting web forms implemented in Java EE or ASP.NET may consist of numerous models, views, web services and databases. It is vital that no crucial information is lost during the process.

- *Improved comprehension* of a system is one of the main goals of MDRE. In order to be able to provide some kind understanding of a complex system to a domain expert or developer, the extracted model that represents the system must be at a higher level of abstraction containing only the most relevant information. For instance, the requirements included in the WCAG 2.0 model focus on the structure and style of the web page, but not the textual contents of paragraph elements or tables. These are the sorts of elements that should be excluded from the abstract view of the web page since they are irrelevant in the current context.

- *Managing scalability* is an important challenge to overcome in order to provide a process that works efficiently. Software systems in the real world are big consisting anywhere between hundreds of thousands to millions of lines of code. Traditionally web pages would

contain considerably less code since most of the logic happened on the server side, but with the emergence of client side frameworks such as AngularJS [16] web pages have grown larger and contain a lot more content than previously. The larger the artefact is the larger the model will be. The MDRE process must be designed so that large models can be manipulated and transformed efficiently.

- *Adapt existing solutions to different needs.* Many of the existing MDRE tools are platform dependent which means that they only support reverse engineering for specific technologies or languages (i.e. CSS or HTML) [12]. Even though it's unlikely that a MDRE solution contains no platform dependent components, steps should still be taken to ensure the solution is as generic as possible.

In addition to the challenges presented above, there is the problem of *Adequate reverse engineering*. The idea of adequacy originates from software testing, where an adequate test set indicates that the software testing phase is complete. The adequacy of a test set is measured by how thoroughly the requirements of an implemented feature is tested. The adequacy criteria for a model in a MDRE process consists of two characteristics: *thoroughness* and *lucidity*. Thoroughness explains how much the artefact is examined by the MDRE process. Lucidity defines to what extent the MDRE process describes how the artefact works and it's purpose. A proposed solution is to run the reverse engineering process twice. Once on the artefact to generate the model, and then once on the generated model. This is done in order to produce a copy of the original artefact. The copy will determine if the model is adequate or not [47].

When attempting to reverse engineer a web page, a possible solution to the adequacy problem could be to generate a HTML document from the result of the MDRE process and determine if the model is adequate by confirming that all the input elements and label elements that occur in the original web page exist in the generated HTML document. The Bottom-up approach will not attempt to do this, but the challenge of model adequacy should be considered for future work.

Requirements

In order to overcome the challenges presented in the previous section, there is a list of requirements a full MDRE approach should satisfy [12]:

- *The Genericity Requirement* - The approach must be technology-independent, this can be achieved by using meta-models. Components that are technology-specific may be plugged into generic components, but steps should be taken to avoid components that are completely technology-specific.

- *The Extensibility Requirement*- The approach must rely on information provided by the model representation of the artefact and not the artefact itself. This is to ensure a clear decoupling between the bottom up approach and the artefact.

- *The Full coverage Requirement* - If necessary the bottom up approach should provide a complete representation of the artefact at different levels of abstraction. This is to minimize the loss of data.
- *The Direct (re)use and integration Requirement* - The MDRE process should consist of as many reusable components as possible, in order to avoid duplicate code or implementing more platform specific components than absolutely necessary.
- *The Facilitated automation Requirement* - Parts of the approach must be automated if possible to improve the efficiency of the bottom up approach. This can be achieved using MDE techniques.

The Meta Model Problem

The MDRE process implies that models play a crucial role in the reverse engineering process, and therefore so do meta-models. It is possible to depict two scenarios: In first scenario the domain in question is familiar enough to be able to create a meta-model that describes the domain sufficiently. In the second scenario the domain is unfamiliar making it difficult to analyse and therefore unable to define a meta-model that sufficiently represents the domain [49].

A proposed solution involves introducing the concept of *Model fragments* [49]. Model fragments are concrete examples of situations from which a meta-model can be defined. These fragments can be defined by a domain expert using a drawing tool or a developer using a textual editor with annotations to guide the induction process.

A developer could for instance create one or more model fragments, using a web page consisting of a html document and corresponding CSS style sheet that satisfies the selected requirements from chapter 2. The model fragments could then be used to generate the WCAG 2.0 model used by the web evaluation tool to evaluate other web pages. Annotations can be used to mark which parts of the web page satisfy certain requirements.

There is a distinction between the two types of annotations [49]: *domain* and *design* annotations. Domain annotations define certain aspects of the model fragment. This can be for instance annotating HTML elements and attributes with *@element* and *@attribute*. These annotations may be used later when compiling the meta-model. Design annotations refer to annotations that decide how certain elements should be represented in the generated meta-model. This could be for instance assigning the *@root* attribute to a form element in order to replace the html-tag as the root element of an HTML document.

The following steps are suggested when defining a new meta-model using Model Fragments, which is illustrated in figure 5.10 [49]:

- Step 1: A domain expert creates one or more example models with the help of a visual sketching tool. These models are then transformed into model fragments which consist of elements and relations between the elements.

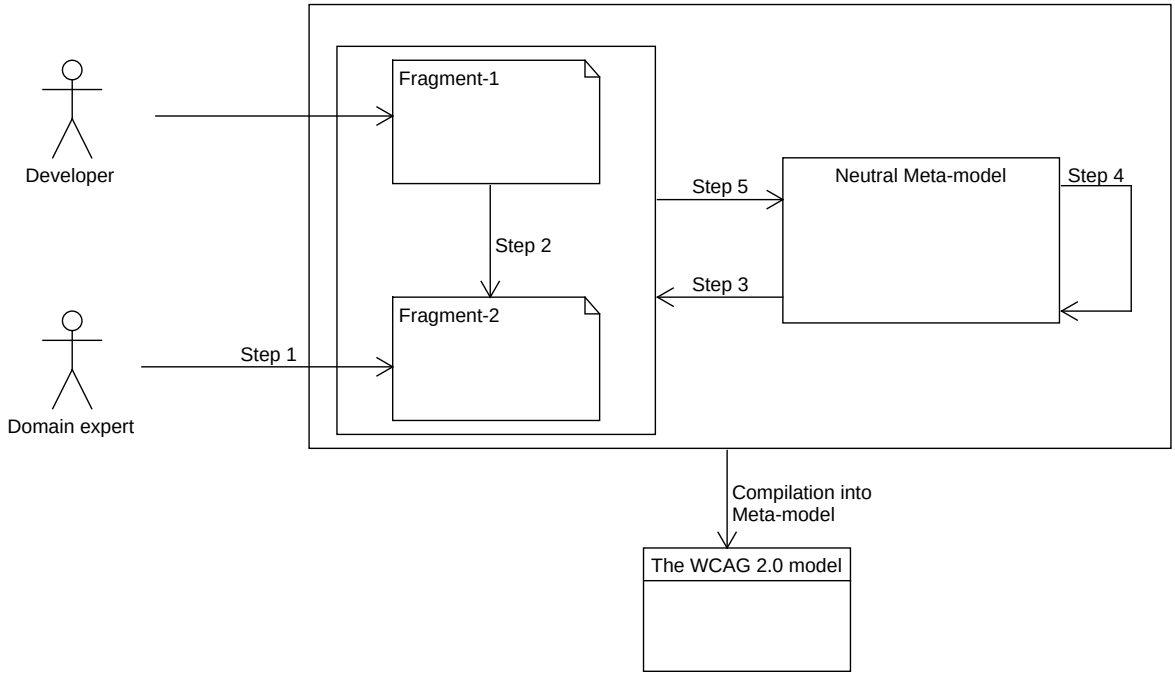


Figure 5.10: Bottom-up meta-modelling with model fragments

- Step 2: A developer can manipulate the fragments and define new ones, as well as add annotations to the fragments to guide the induction process.
- Step 3: A meta-model is generated from the fragments and their annotations.
- Step 4: The meta-model can continue to evolve by either adding new model fragments and regenerating the meta-model as done in step 3, or by manually refactoring the meta-model
- Step 5: A checking procedure detects possible conformance issues between the new meta-model and the existing fragments.

The meta-model is extracted from the model fragments using *The Meta-Model Induction Algorithm* [49]. The Meta-Model Induction Algorithm is a technique, which when given a model fragment will parse it and for each object with a distinct type will create a new meta-class in the meta-model. The meta-class is only added if it doesn't exist in the meta-model. The same is done for the for each attribute and relation originating from the object. If multiple relations with the same name targets objects of different types, an abstract super class is created as the target of the relation and sub class is created for each object.

So far there's been presented two different approaches to defining the meta models used in the reverse engineering process: The developer designs the meta model by hand, or that the developer uses the Meta-model Induction Algorithm to generate a meta model from model

fragments. There is however a third approach which involves using a standardized generic meta model called the Knowledge Discovery Meta-Model which was described in section 3.1.3 [41].

5.2.2 The Model-Based Reverse Engineering Approach

The Model-Based Reverse Engineering process can be broken down into two phases. *The Model Discovery Phase*, where a model is extracted from the artefact, and *The Model Understanding Phase* where the extracted model is transformed into an abstract view of the system. The MDRE process can be defined as the following[12]:

Model Driven Reverse Engineering = Model Discovery + Model Understanding

These phases are not affected by how the meta-model is created, the MDRE process is executed in an identical manner, regardless of if the meta-model is generated from a model fragment or if the meta-model was manually created by a domain expert [12].

The Model Discovery Phase

During the Model Discovery phase a raw model is extracted from the artefact which is to be reversed engineered. These extracted raw models are referred to as initial models and should be extracted as early as possible during the MDRE process in order to avoid losing any essential information. By doing so it is possible to move the problem from the technology dependent technical space to the model driven conceptual space from the start and not be concerned with components suited only for a specific technology. The goal is make the components in this phase as generic as possible so that it may be reused frequently [12].

Once the initial model has been extracted from the artefact, it is possible to interact and manipulate the model using MDE principles and techniques. The initial model extracted has to be sufficiently accurate, allowing the process to use it as a starting point for the next phase. The initial models are usually considered to be models of low abstraction, which means they do not provide any significant increase in the level of abstraction or detail. This is to ensure that the initial model has lost a minimal amount of information during extraction [12].

In order to identify and extract the initial model, a component referred to as the *Model Discoverer* is used to generate the initial model from an artefact. The Model Discoverer is generally designed to be suited to a specific technology or language such as CSS or HTML in order to extract the initial model correctly and as complete as possible. The Model Discoverer generally uses a two step process when extracting the initial model: The *Injection* step and the *Transformation* step, see figure 5.12.

The Injection step involves using parsers, such as *Delimiter-directed Translation* or the *Syntax-Directed Translation* to extract the information belonging to the artefact in order to create the

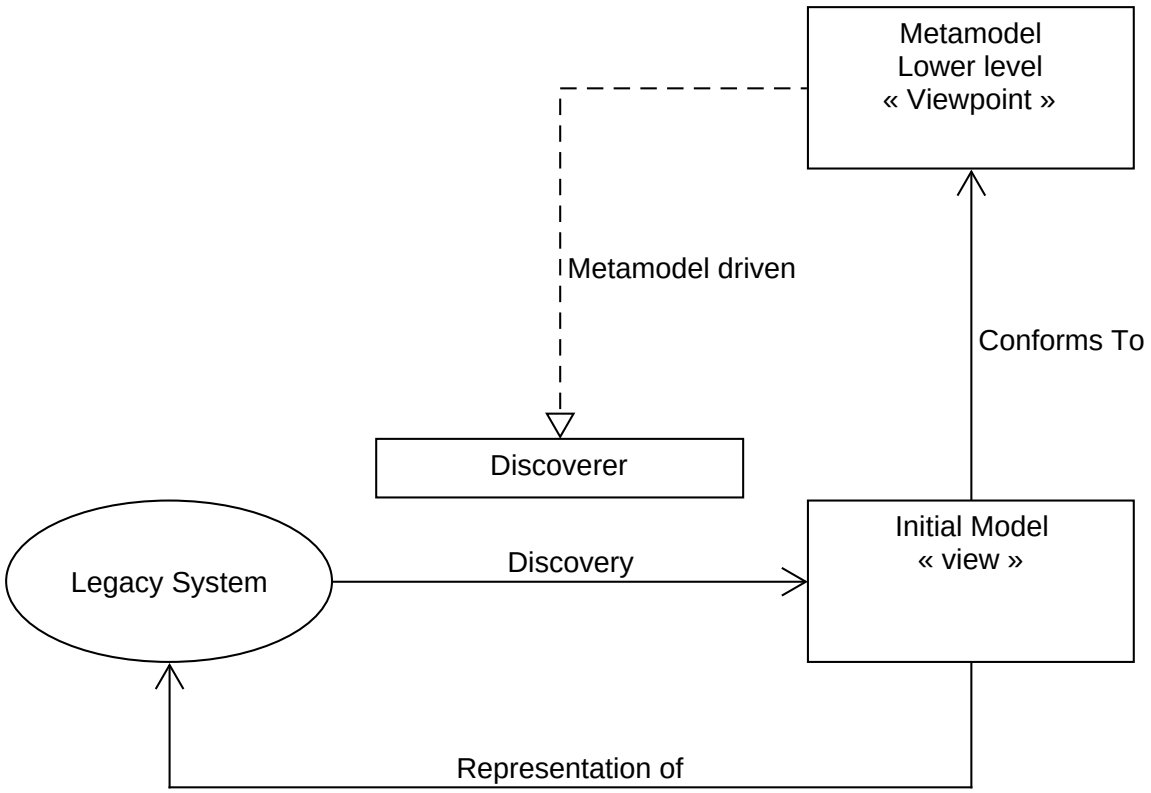


Figure 5.11: The Model Discovery phase

initial model. Alternatively APIs provided by the artefact can also be used. Basically this step focuses on bridging the technical space where the artefact exists and the conceptual space where the initial model is created [12].

The transformation step provides an additional mapping within the conceptual space that uses the result from the injection step to generate the initial model. Considering the case presented in chapter 1, the Injection Step would consist of extracting an XML representation of the HTML document, that can be more easily traversed and manipulated. The Transformation Step would then involve transforming the XML to create models that can be used in the evaluation process. The result of the Transformation Step is the initial model which is correctly typed by the meta-model defined prior to this phase. Figure 5.12 shows an example of the Model Discovery Phase [12].

Note that it is not always necessary to extract the artefact to some generic model such as XML before transforming it to the initial model (as described in the previous paragraph), in some situations the model discoverer might have access to all the information it needs from the artefact or that information is adequately structured, and can therefore take a more direct approach by building the models directly. These two approaches are referred to as: The *Two-step approach* and the *direct approach*. The Two-Step approach involves extracting the artefact to a

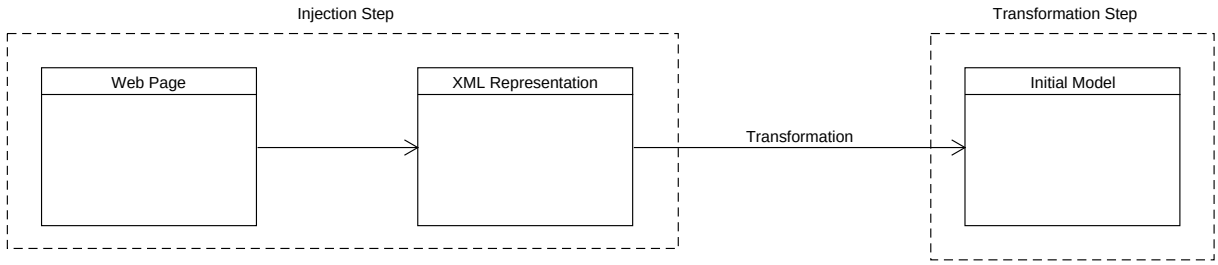


Figure 5.12: The Two-Step Approach

generic model followed by transforming the generic model to the initial model. This approach involves more steps, but each step is less complex and the Injector is generic and reusable. The Direct approach uses the available information in the artefact to build the initial model directly. In this approach the transformation step is integrated into the injection step, which means less, but more complex steps. There are no concrete rules when to use a Direct approach or Two-Step approach, and ultimately the result is the same: an initial model that serves as the starting point of the next phase [12].

When considering the web page domain, the direct approach might be considered adequate when extracting the initial model from the html document, whilst the two step approach might be considered the better choice when extracting the initial model from the CSS document (First extract CSS rules to a key value data structure, then map it to a XML). Figure 5.12 demonstrates the Model Discovery Phase.

The Model Understanding Phase

During the Model Understanding phase, chains of model manipulation techniques are used to query and transform the initial model in order to obtain a model of the system at a higher level of abstraction. The result of this phase is a model commonly referred to as the *Derived Model* [10].

The Model Understanding phase consists of four operations executed within a iterative model transformation process [10]:

1. *Model Navigation* is the operation of exploring the reversed engineered artefact using the extracted initial model instead of exploring the artefact directly.

2. *Model Querying* is the operation of filtering out the irrelevant results from the Model Navigation operation so that only the information needed by the Model Computation operation is preserved.

3. *Model Computation* is the operation where the results from the Model Querying are used to compute the derived model which is built in the last step of the process.

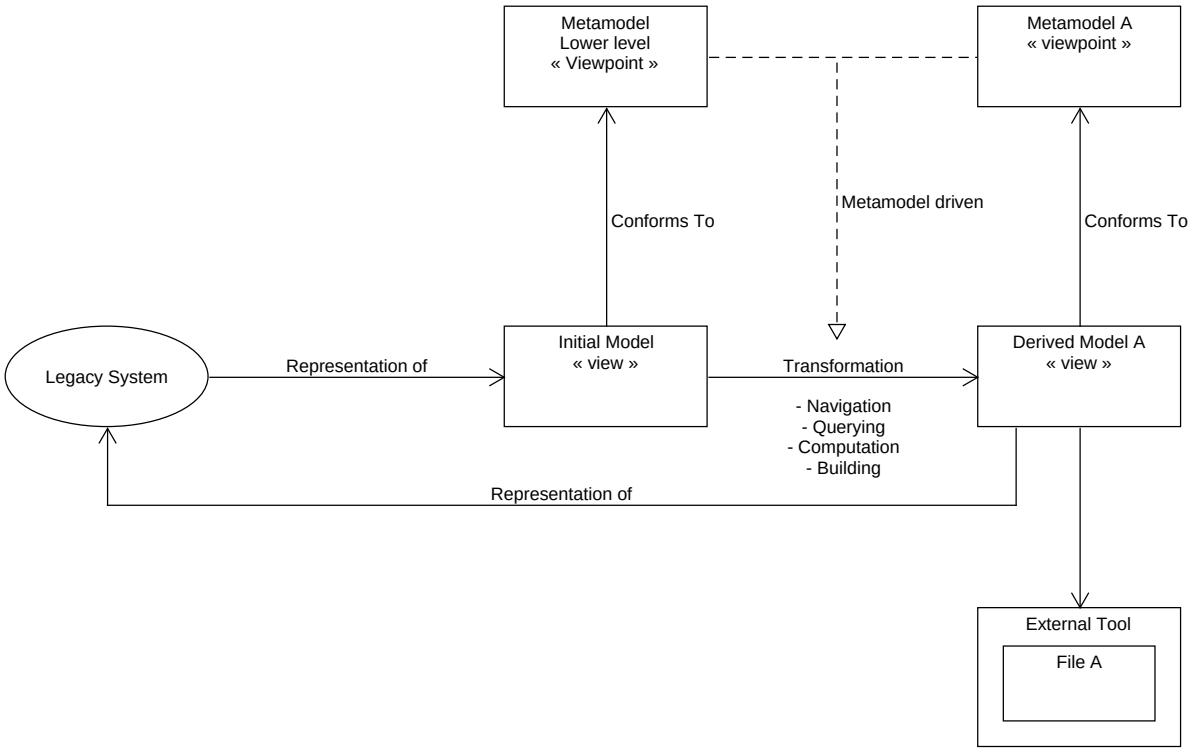


Figure 5.13: The Model Understanding phase

4. *Model building* is the final operation where the derived model is generated based on the computed model from the last operation.

This phase results in a derived model that can be exported to an external tool for further use. Figure 5.13 depicts the Model Understanding phase.

Model (Re)Generation

In some scenarios the MDRE process is part of a larger reengineering process (see figure 3.4), in those cases a third phase is included in the MDRE process. The Model (Re)Generation phase involves generating the expected outcome from the derived model (e.g. a piece of executable code). Since the goal of this approach is to validate the abstract model of an artefact and not generate a runnable output, this phase will not be included in the Bottom-Up approach [10].

5.2.3 The Template-Based Reverse Engineering Approach

An alternative to the MDRE approach is the *Template-Based Reverse Engineering (TBRE)* Approach [7]. TBRE is an approach that goes hand-in-hand with code generation. One of the main

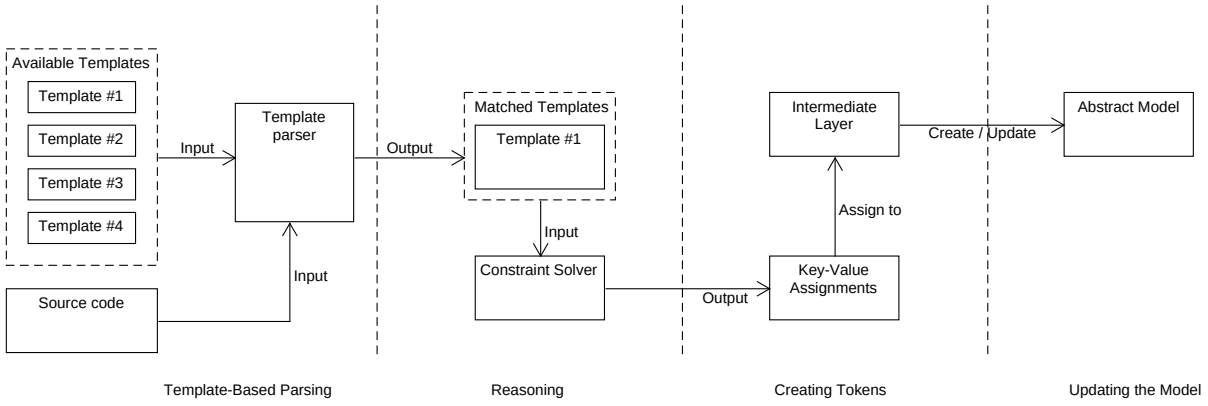


Figure 5.14: Depiction of Template-Based Reverse Engineering Approach

challenges with code generation is minimizing the gap between the meta-model and the generated source code. When a developer makes manual changes to the generated source code, the model becomes an incomplete representation of that source code, thus a gap is created. The Generation Gap technique is one way of solving this problem, an alternative is using a *round-trip engineering tool*, which reflects changes made to the source code back to the model in order to keep both in sync. The TBRE approach attempts to do this using the same template used for generating the code from the model, this means that a template based code generator is a prerequisite when using this approach.

When Reverse Engineering a software artefact it's common practice to use a parser designed for the target language for the software artefact. For instance, even though the MDRE approach attempts to create a solution that is as generic as possible, it still requires a model discoverer, which are normally platform specific [12]. There are several disadvantages with this: First of all, a separate parser must be created for each target language, one could argue if the process is truly generic if it can only reverse engineer certain target languages. Second of all, the result of parsing a piece of software is often fragile and rigid, therefore mapping the result to a model in the solution can be quite laborious [7].

TBRE recognizes this problem, and although it still requires a mapping from the software artefact to the model, it is easier to work with since the result from the template-based parser is generally a higher abstraction level than the traditional language parser. Since the reverse engineering mechanism and the code generator mechanism use the same template, whenever the template is changed by the developer, both mechanisms are affected. This solves a major part of the maintenance problem [7].

The TBRE approach is divided into four steps: *Template-Based Parsing*, *Reasoning*, *Creating tokens* and *Updating of the Model*.

Template-Based Parsing

The first step involves determining whether the selected template was used to generate the source code that is to be reverse engineered. This is achieved by first deducing which values were assigned to the template when generating the source code. The template is traversed by a parser that attempts to match the static text in the template with text fragments from the source code, any variable the parser comes across is stored with the initial value "pending". When the parser comes across a branching statement, which is basically a statement that determines what should be generated based on some condition, the parser checks both possibilities by first parsing the section with the condition set to true, then again where the condition is set to false [7].

Once the template has been traversed by the parser, it's time to assign values to the stored variables. Trying every combination hoping for a match can become a very time consuming task, therefore it is possible to specify constraints for the different variables in the template, thus excluding values that are guaranteed to not match the source code. The constraints may vary from simple ones that exclude variables containing white spaces, to complex constraints using regular expressions to determine if the value is valid [7].

Reasoning

The previous step often results in multiple solutions for how the variables can be assigned to the template. It is the goal of the reasoning step to identify contradicting results and remove them. The first step consists of using a *constraint solver* which assigns a boolean value to an unassigned variable in a branching condition where ever possible. The constraint solver can determine if the variable is true or false by checking if the contents of the branching statement are found in the source code, if the source code does not contain the contents of the branching statement, the branching condition must return false when evaluated, if the contents do exist the branching condition must return true [7].

Since the assigned variables in the branching condition may be recurring throughout the template, it must be ensured that the assignment is consistent. This is done by keeping track of how many times the variable assigned a value. If it's only assigned once it can be classified as a read-only variable, if it's assigned multiple times it's classified as a mutable variable and may not be used in the reasoning step. Should any contradictory assignments occur, i.e. the contents of a branching statement exists in the source code, but shouldn't according to the current solution, that solution can be safely dismissed [7].

Creating tokens

The reasoning step provides a set of key-value assignments for each pair of template and source code. These sets of values are assigned to a generic model referred to as the intermediate layer,

this way updating or creating the meta-model becomes a model-to-model mapping, making the process reasonably simpler [7].

The goal of this step is to create the intermediate layer consisting of a token-tree, referred to as the token layer, that the key-values from the previous step are assigned to. Tokens in the intermediate layer function similarly to the context layer in the templated generation described earlier in this chapter. The creation of the token-layer is done with the following steps [7]:

1. For each paired template and source code, and their corresponding assignments, referred to as a solution, a token object is created and added to the token-layer.
2. The textual assignments are mapped to their corresponding attributes in the created token.

At the end of this step, all textual assignments should be mapped to a token in the token-layer.

Updating of the Model

The final step is to create or update the model, which basically happens by traversing the token-layer and modifying the model based on the contents of the tokens [7].

Up until now, the TBRE process has been generic, but in this step it is necessary to include components that are platform dependent. This is because it is not possible to create a generic mapping function that can adequately map the values in the tokens to the model, even if the model is empty. The biggest challenge is to ensure that the created model is a sufficient abstraction of the artefact, otherwise there is the risk of the model containing a lot of irrelevant information. This why the final step needs a mapper that is specified for a certain meta-model [7].

The result of this step is an abstract model of the artefact that was reverse engineered.

5.2.4 The Bottom-up Model Validation Process

This section will explain *The Bottom-up Model Validation Process*. It will first justify the method of obtaining a meta-model, followed by a short discussion on which reverse engineering approach is best suited.

The first aspect that should be noted about this approach, is that as opposed to the top-down model validation approach, the Bottom-Up approach attempts to move the problem from the technical space to the conceptual space. This is achieved by extracting an abstract model of the web page and evaluating it by checking if it conforms to the WCAG 2.0 model. Should the

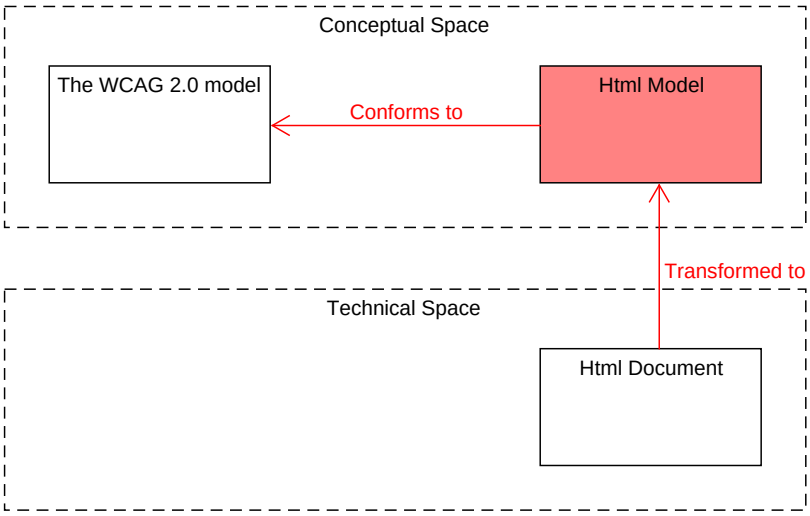


Figure 5.15: Conceptual figure of the Bottom-up Model Validation approach

abstract model successfully conform to the meta-model, the web page can be considered valid. Should it fail to satisfy one or more constraints, the model does not conform to the WCAG 2.0 model, and thus the web page is invalid.

The second aspect that should be discussed is how the meta-model is obtained. Considering the discussion earlier in this chapter, there are three methods of acquiring a meta-model that can be used in the Bottom-up approach. The first method is designing the meta-model manually with the aid from a domain expert. The advantage here is that a common understanding of the requirements, regardless of competence, is achieved. This means it is possible for the domain expert to sign off on the model, assuring that the requirements have been correctly interpreted and implemented. Moreover, it is easier to ensure that only relevant information is included in the model, keeping both the model and the corresponding meta-model tidy and easy to understand. Finally by designing the meta-model manually with the domain expert, the model and the meta-model will be correctly defined using a common DSML based on the current domain.

The second method is using model fragments and the Meta-model induction algorithm. This alternative is best suited for situations where the domain is unfamiliar, or there is no domain expert. By creating examples of web pages that satisfy the requirements specified by WCAG 2.0, it is possible to automate the generation of a meta-model. However, there are several drawbacks. First of all by automating the generation of the meta-model, it is more difficult to ensure that the meta-model achieves the adequate level of abstraction or that it describes the domain correctly. Secondly, as explained in section 5.1.1, generated code tends to be more difficult to understand, the same problem could be applied to a generated meta-model. The meta-model might loose its important function of describing the requirements in a way that can be interpreted by both the domain expert and the developer.

The third method is using the generic meta-model KDM to capture all relevant informa-

tion from the web page. Considering the domain described in chapter 1 and the challenges presented in chapter 2, the Knowledge Discovery Model is a bit excessive. Mainly because the Bottom-Up approach treats the web page as a class 1 static web page and not a class 3 web page which relies heavily upon server side logic.

Considering the arguments presented for each of the three methods for obtaining a meta-model, the first method involving a meta-model designed with a domain-expert is the most preferable for the Bottom-Up approach.

The third aspect is deciding whether the TBRE or the MDRE approach should be implemented in this process. Even though the TBRE approach provides many advantages such as simplifying the mapping from the artefact to the model, and keeping the model in sync with the artefact, none of the benefits with the TBRE approach are particularly applicable to what Bottom-Up approach aims to achieve. Furthermore, these benefits come with a price since the TBRE approach requires a template that can be combined with the artefact (in this case a web page), in order to generate the abstract model. When considering these aspects, the MDRE approach appears to be better suited than the TBRE approach, therefore the Bottom-Up approach will consist of the MDRE approach.

The process

The Bottom-Up approach consists of three steps that are executed as follows:

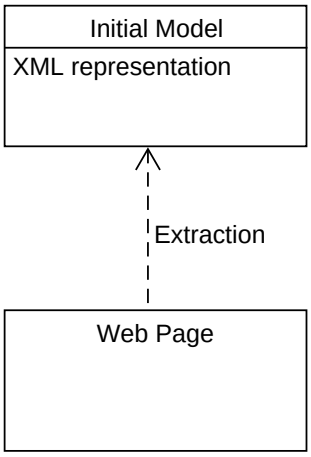


Figure 5.16: Step one: Extract a initial model of the Web Page

During step one, an initial model is extracted from the web page. This could be a simple mapping such as a HTML-to-XML achieved using a Html Model Discoverer, or from CSS-to-XML using a CSS Model Discoverer. The most important thing to keep in mind is that the extracted initial model is as detailed as possible in order to preserve all the information the web page contains. See figure 5.16.

Looking back at the *1.1.1 Non-text Content* requirement example from the Top-Down approach, the web evaluation tool needs to be able to ensure that every html image element contains the alt-attribute. Here the html image element was evaluated using some piece of generated logic, for instance an XQuery function that looped through every image element and returned the element if it was missing the alt-attribute. In the Bottom-Up approach, the web evaluation tool is attempting to confirm that the model is correct instead of the actual web page. This means the web evaluation tool can utilize methods and techniques provided by MDE.

In step one with the aid of a html interpreter, the web page is parsed and a initial model represented as an XML is extracted from it. The initial model contains all the information contained within the web page, including the html image elements that are to be validated.

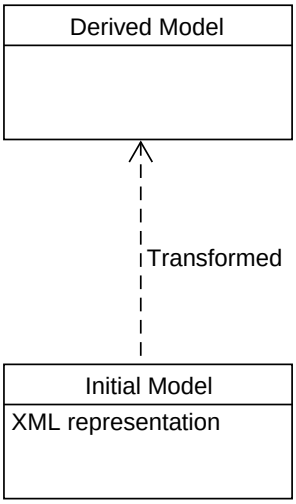


Figure 5.17: Step two: Transform the initial model into the derived model

The second step consists of an exogenous model-to-model transformation where the initial model extracted in the first step is transformed to the derived model which in this case is an instance of the $\mathfrak{S}_1$ specification in the WCAG 2.0 model, see section 4.3.2. Technically the derived model could be considered as the $\mathfrak{S}_2$ specification in the modelling hierarchy described in chapter 4. The derived model is generally more abstract containing only the necessary information, for instance none of the selected requirements from chapter 2 focus on any potential html tables the web page might contain, therefore any html table included in the web page can be excluded from the derived model.

Continuing with the example from the previous step, the html image elements contained in the initial model are mapped to an instance of the WCAG 2.0 model. They are no longer structured as an html element with attributes, but rather a set of nodes and edges representing the html element and it's attributes with a graph-based graphical syntax. Irrelevant information such as user events attached to the element are not included in the transformation, ensuring that the new representation of the html image element only contains the information that is necessary in order to confirm that it satisfies the *1.1.1 Non-text Content* requirement.

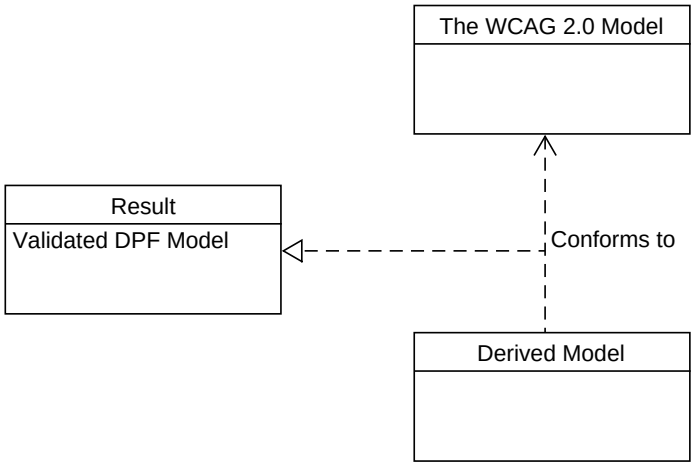


Figure 5.18: Step three: Check if derived model conforms to the WCAG 2.0 Model

The final step in the Bottom-Up approach consists of checking if the derived model successfully conforms to the WCAG 2.0 Model, if any parts of the derived model are incorrectly typed, or don't satisfy the corresponding predicate, the node is flagged as invalid.

Concluding the example from the two previous steps, any html image element that is missing the alt-attribute will not conform to the WCAG 2.0 model reporting that the multiplicity predicate is not satisfied.

Figure 5.19 displays an example of a valid web page based on figure 2.1. This derived model is considered valid because it contains the required *for* and *labelledBy* edges between the *label* node and the *input* node. Figure 5.20 shows an example of an invalid web page based on the figure 2.2 where the *input* node is missing both the required edge to a *label* node, and the alternative *attr* edge to a *title* node.

5.3 Summary

Essentially it is where the evaluation process is conducted that differentiates the Top-Down approach from the Bottom-Up approach. In the Top-Down approach, the web page is evaluated in the platform dependent technical space, whilst in the Bottom Up approach attempts to evaluate the web page in the platform independent conceptual space. This section will compare the two approaches by addressing some of the aspects that differentiates them, as well as some of the challenges presented in chapter 2. See table 5.1 for a comparison of the two approaches.

The first aspect to discuss is evaluating the web page in the conceptual space versus the technical space. The advantage of generating code that can evaluate the web page in the technical space is that evaluation process gains access to framework specific code and available APIs.

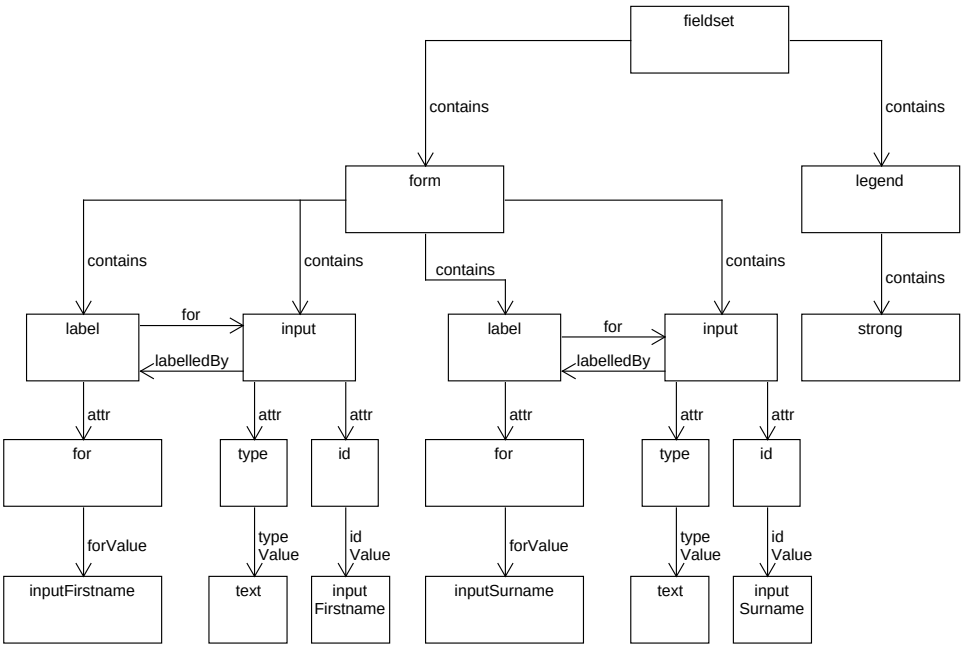


Figure 5.19: Example of a valid derived model extracted from the validated web page

This means that it is possible to create complex procedures for evaluating a web page, thus providing accurate results. However, by choosing to solve the problem in the technical space, the evaluation process becomes obscure to the domain expert, making it difficult to determine if evaluation process is correct.

The second aspect that should be considered is how the results from the evaluation process are presented. The Top-Down approach outputs a textual report, while the Bottom-Up approach returns a graph-based model. One of the challenges with generating a textual error message is ensuring that it is detailed enough to be useful to the domain expert or the developer. It is not necessarily sufficient to pinpoint which elements are invalid, often the domain expert will want to know the cause of the violation is in order to fix it, this is addressed in *the specificity problem*. By presenting the error message using a graph-based model, it could be possible to reflect the violation much more effectively than a generated textual description does. Perhaps the best way to describe this approach is with the English idiom "*A picture is worth a thousand words*".

Approach	**Space**	**Process**	**Result**	**Transparent Evaluation**
Top-Down	Technical Space	Platform specific functions	Textual	No
Bottom-Up	Conceptual Space	Model Conformance	Graph-based model	Yes

Table 5.1: Comparison of the Top-Down Approach versus the Bottom-Up Approach

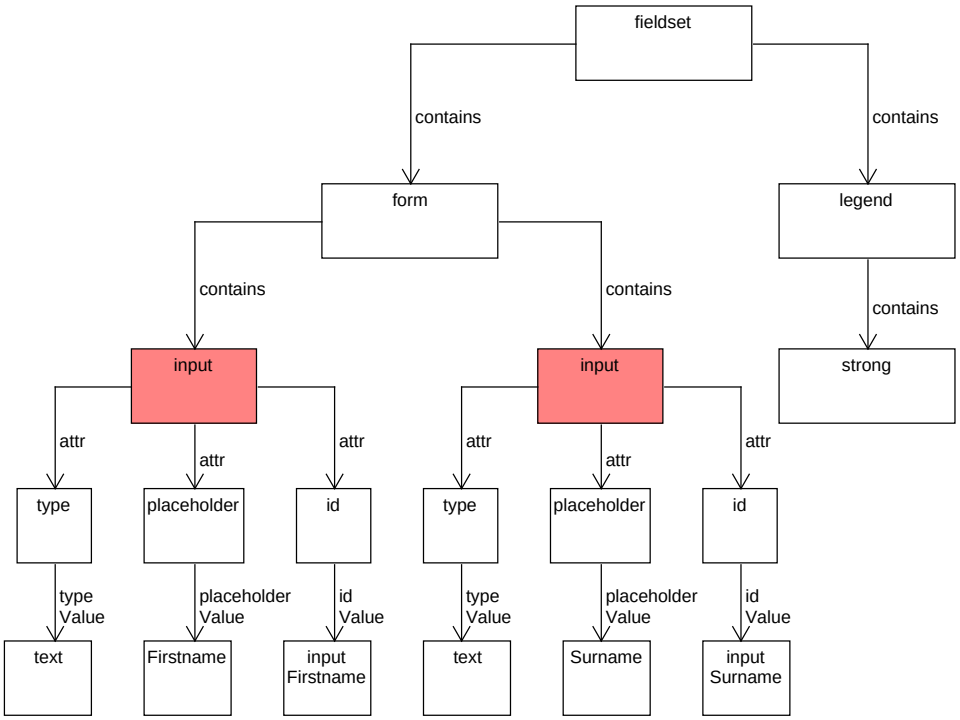


Figure 5.20: Example of an invalid derived model extracted from the validated web page

Both aspects play a crucial part in providing an adequate solution to *the transparency problem*. As explained above, by solving the problem entirely in the technical space, the evaluation process is hidden from the domain expert. This means it can be concluded that the Top-Down approach does not solve the transparency problem. The evaluation process in the Bottom-Up approach is executed in the conceptual space by using conformance checks, and therefore could be considered as a valid solution for the transparency problem.

In addition to the transparency problem, there are a few other challenges the evaluation process should overcome. For starters, there is the *The Modification Problem*, which both approaches provide a solution for, by ensuring that all changes made to the requirements are confined to the WCAG 2.0 model. In the Top-Down Approach, changes to the model are reflected in the generated validation logic, thus changing how the web page is evaluated.

Looking back at the example concerning the *1.1.1 Non-text Content* requirement, should for instance an additional requirement be added stating that every html image element requires an ID. This requirement could be enforced by adding a new multiplicity constraint to the WCAG 2.0 model, this constraint would then be enforced using a generated function that will loop through every image element and identify any element missing the ID attribute. However, changes made to the WCAG 2.0 model will require a new code generation process in order to update the validation logic.

It could be argued that the Bottom Up Model Validation approach provides a more elegant

solution to the modification problem, since no new source code needs to be generated. Worst case scenario, the model representation of the web page is inadequate and changes to the model transformation between initial model to the derived model needs to be adjusted. Should the model be adequate, any changes to the WCAG 2.0 model will be included during the conformance check.

The Generic Evaluation Problem involves ensuring that the evaluation process is reusable, in order to be able to evaluate the multiple components in the a web page. The Top-Down approach generates validation logic that targets a certain artefact such as Html. This means that whatever code is generated to evaluate a Html document, will most likely not be reusable for evaluating other components such as CSS documents. This could also potentially mean that separate models need to be designed for each component belonging to the web page.

The Bottom-Up approach uses model discoverers to extract initial models of the target artefact. The derived model that the initial model is mapped to is platform independent and generic in the sense that it can represent any component belonging to a web page. Furthermore the conformance check conducted between the WCAG 2.0 model and the derived model is also platform independent. The evaluation process described in the Bottom-Up approach can therefore be considered a generic, and thus provide a possible solution to the Generic Evaluation Problem. It should be noted, that even though the derived model and the evaluation process are platform independent, a platform specific model discoverer still needs to be implemented in order to extract the initial model

The Ambiguity Problem focuses on the need for a common understanding between the developer and the domain expert, with regards to how the requirements should be implemented. This could be considered one of the most important aspects of the solution presented in this thesis, since many potential problems may occur because of misunderstandings when interpreting the requirements or miscommunication between the domain expert and the developer. Since the WCAG 2.0 model that both approaches rely on, provides a clear and concise description of the mandatory requirements a web page must fulfil, any potential misunderstandings between the developer and domain expert are prevented.

In the light of the transparency problem, the Bottom-Up approach has been selected as the preferred solution. The next chapter will discuss how well this solution handles the requirements presented in chapter 2 as well as compare the implemented solution with other existing web evaluation tools.

Chapter 6

Evaluation

In this chapter the solution, which consists of the WCAG 2.0 model described in chapter 4 and the Bottom-Up approach from chapter 5 will be evaluated. The evaluation will be conducted based on the challenges presented in chapter 2, a conducted usability experiment and by measuring its ability to detect errors in a web page.

In order to determine how well the WCAG 2.0 model solves the Ambiguity Problem, a small usability experiment has been conducted. The participants of the experiment consisted of a developer and a consultant from the Norwegian software company ACOS [2], which specializes in developing software for the Norwegian government and municipalities. The company develops several large software systems within document management, health-care, and web services. The participants from ACOS have been involved in previous governmental web based projects that required parts of the project to conform to the standards set by DIFI. The results from the usability experiment will be discussed in section 6.1.

Furthermore, The Bottom Up Approach has been implemented as a plug in for the WebDPF tool [36], which is a web based editor created at the University College of Bergen, used for designing and working with DPF models. This has been done in order to be able to evaluate The Bottom Up Approach as a web accessibility evaluation tool.

Finally, as mentioned in chapter 1, by choosing to focus solely on the mandatory Universal Design requirements specified by DIFI, the Bottom Up Approach can be classified as a Web Accessibility Evaluation tool. Thus the effectiveness of the Bottom Up approach can be assessed by comparing it to other state-of-the-art Web Accessibility Evaluation tools. This thesis has selected the following tools to compare the bottom-up approach with: *AChecker* [1], *SortSite 5*[44] and *Wave*[59]. This chapter will conclude with a summary that will highlight certain aspects addressed in this chapter.

6.1 Usability Experiment

One of the most important challenges that has been referred to in this paper, is the ability to close the gap between developer and the domain expert, and in doing so providing the means to allow the two to reach a common understanding of the requirements that a web page must fulfil. Achieving a common understanding helps prevent challenges caused by the ambiguity problem. This is one of the primary goals of the WCAG 2.0 model designed in chapter 4. In this section the findings from a conducted Usability Experiment consisting of a small group of participants will be presented.

The purpose of this usability experiment is to investigate and determine if the WCAG 2.0 model from chapter 4 is able to provide a clear and concise channel of communication between a developer and a domain expert. It should be noted that because of the size of the experiment, the results should be considered as indicative, and not as something conclusive. A detailed report from the conducted experiment can be found in appendix D.

6.1.1 Setting up the Experiment

The usability experiment was conducted in one of the seminar rooms at the University College of Bergen, and consisted of three individuals: A domain expert from Acos, a developer from Acos and an observer. All information was provided to the participants electronically using a television screen to display each model. Furthermore, the participants were provided several sheets of paper for sketching their own models.

The experiment started off with a short explanation of how the WCAG 2.0 model is to be interpreted, then it was divided into the following three parts:

- **Interpreting existing models** - In this part the participants were asked to describe which requirements each model contained.
- **Understanding model conformance** - The participants were shown several meta-models with corresponding models that did not successfully conform. They were asked to identify the cause and provide a solution to correct the model.
- **Designing new requirements** - The participants are given a set of criteria that they must attempt to model.

The data collected from parts one and two were gathered by direct observation conducted by the observer, and can be found in section E.1 and section E.2. The data from part three was registered by having the participants sketch down their answers on the paper provided to them, this can be found in section E.3.

Once the observer had provided a short explanation of the WCAG 2.0 model, he was more or less silent during entire usability experiment in order to allow the participants to reach their own conclusions without input from someone familiar with MDE, DPF and the WCAG 2.0 model.

6.1.2 Interpreting Existing Models

The first part consisted of showing the participants nine different DPF models. The participants had to identify which requirements each model contained. The goal with this part was to determine if both the domain expert and the developer were able to understand the contents of the meta-model, and in doing so, support the idea that the WCAG 2.0 model might be able to provide a common understanding of the requirements.

Assignment	Answer From Paricipants	Correct
1.1 (A.1)	Every image element needs to contain exactly one alt attribute	Yes
1.2 (A.2)	No DIV element may contain a Html element that is incorrectly implemented	Yes
1.3 (A.3)	An Input element must either have the title-attribute, be labelled by a label element or both.	Yes
1.4 (A.4)	Every header must appear sequentially, a H3 element may not be implemented without a H2 element appearing before it, and a H2 elementmay not be implemented with a H1 element appearing before. The H1 element is not dependent on any other header.	Yes
1.5 (A.5)	A label must either contain an input element or refer to it using the for attribute, or both.	Yes
1.6 (A.6)	Every the background colour and the foreground colour of every label element must have a contrast of 4.5:1.	Yes
1.7 (A.7)	A form element must contain at least one label element and one input element. Each label element must contain exactly one input element. Each input element must contain either a title attribute, or be labelled by a label element.	Yes
1.8 (A.8)	A hyperlink element may contain a H1-element, but not vice versa	Yes
1.9 (A.9)	The Html element must contain the lang-attribute.	Yes

Table 6.1: Usability Experiment: Results from part 1

Table 6.1 shows the results from part 1, and it would seem that the participants were able to successfully identify the requirements in each of the models presented to them. It is possible to draw the conclusion that the WCAG 2.0 model could serve as a suitable medium for communicating web page requirements between individuals with different areas of expertise. Furthermore, this can also indicate that the WCAG 2.0 model has fulfilled it's intended purpose in providing a clear and concise description of the requirements.

6.1.3 Understanding Model Conformance

The second part involves showing the participants several meta-models and their corresponding models that fail to conform. The purpose of this exercise was to determine if the feedback from the web evaluation tool, could be understood by the participants. The idea is that by presenting the error visually as a graph, the user will be able to determine not only what is wrong, but also what the cause of the problem is and provide a suitable solution.

Assignment	Answer From Participants	Correct
2.1 (A.1)	In order for the model to conform to the metamodel, the IMG element needs to refer to an ALT attribute using the attr edge	Yes
2.2 (A.2)	In order for the model to conform, the h1 element must not contain the a element.	Yes
2.3 (A.3)	Each TR element must contain at least on TD element, the failed TR element needs to refer to at least one TD element in order to be valid. The theader was invalid because it contained a TD element, and not a TH element. The theader element could be fixed by changing the headercelle from a TD to a TH. Finally in order for the Table element to be valid, a caption element needs to be added, or the summary attribute.	Yes
2.4 (A.4)	Change the H2 element to a H1 element, the H3 elements to H2 elements, and the H5 element to a H3 element.	Yes

Table 6.2: Usability Experiment: Results from part 2

Table 6.2 shows that the participants were able to easily identify which elements had been flagged as invalid, and with the help of the meta-model, they were able to determine the cause of the error.

Assignment 2.4 (see figure 6.1) presented an interesting situation, the error showed that the order of the headings was incorrect. There are several possible solutions for correcting the order: removing the invalid headers, or adding the missing headers so that the order the headers appear sequentially. Neither of these solution might be suitable however. When viewing the derived model the participants were able to deduce that each DIV element contains a header, and that the most appropriate correction would be change all the existing headers, including the h3 headers that were not flagged as invalid, so that they appear in a sequential order. In other words, the root cause of the violation was not that there were any missing headers, but rather that the web page contained wrong level headings to begin with, and that needed to be replaced with headings at the proper level. This is a problem with a root cause that might be difficult to detect and describe sufficiently with words, but easy to show with a graph-based model.

The results from part 2 indicate that the participants were able to determine which elements were invalid, why they were flagged as invalid and how to correct them in the most appropriate way.

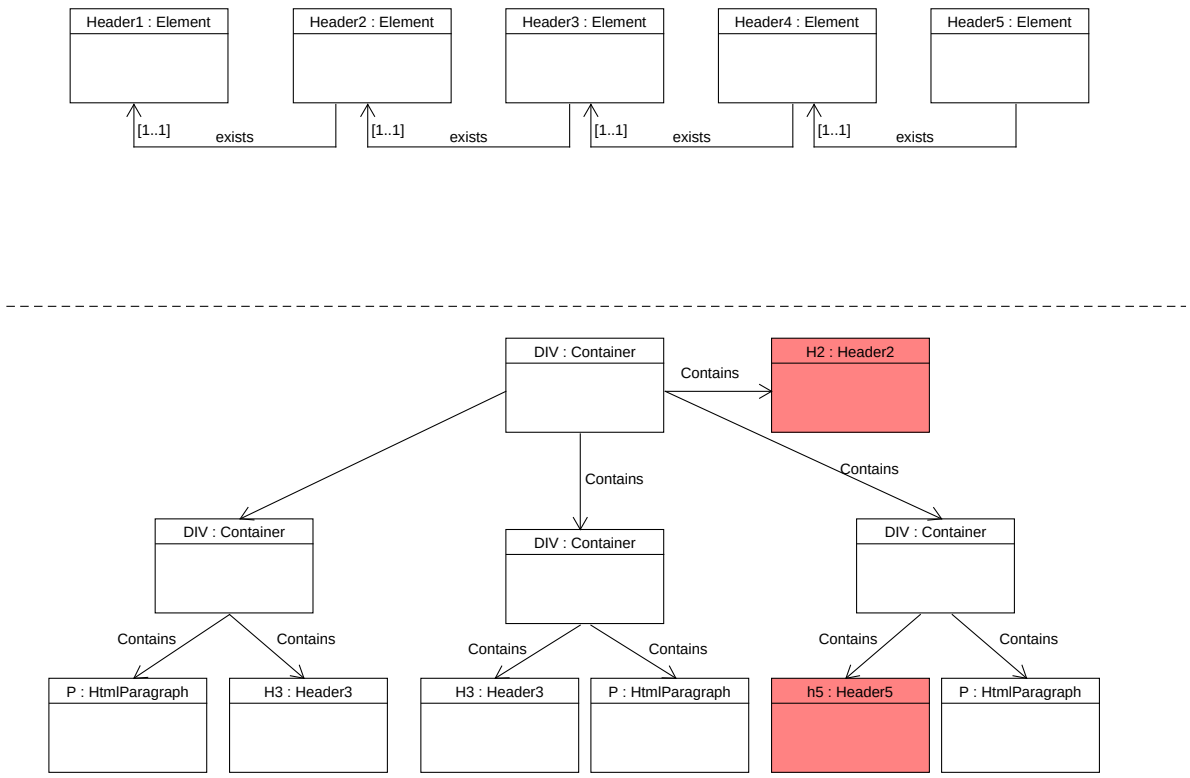


Figure 6.1: Model of a web page that does not conform to it's meta-model

6.1.4 Designing New Requirements

In part 3 the participants were asked to design a model for each of the eight requirements provided to them in a natural language. The purpose of this part was to check if the participants were able to not only understand the requirements in a model, but to be able to make changes to existing models in order to modify current requirements or to include new ones.

The results presented in table 6.3 show that the participants were able to create models that correctly represented most of the requirements. There were however a few models that were not defined entirely correctly.

The most interesting result from this part of the usability experiment was the solution to assignment *3.6 - All non-text content that is presented to the user has a text alternative that serves the equivalent purpose.* The requirement to be modelled includes a barrier where a person needs to determine if the alternative text provides an adequate description of the non-textual content. Instead of ignoring the barrier such as the Bottom-Up approach does, the participants introduced a new edge between the Non-Textual node and the alternative text node named *reason*. The participants explained that the *reason*-edge represents a need for human interaction. This seems like a rational solution to the barrier problem, however it might of been better to represent this demand for human interaction using a notation instead of an edge, see figure 6.2.

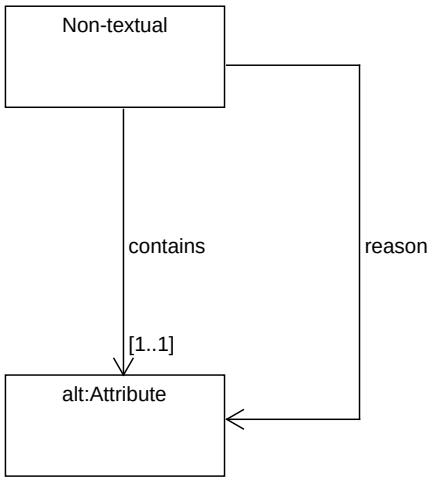


Figure 6.2: Model designed by participants, the edge labelled as *reason* indicates that some sort of human interaction is required

Assignment	**Answer From Paricipants**	**Correct**
3.1 - A table cell may not contain any input elements.	Figure (A.1)	Yes
3.2 - A label may not contain more than input element	Figure (A.2)	Yes
3.3 - A label element may not contain an input element, and refer to an input element.	Figure (A.3)	Yes
3.4 - An Input element must be referred to by a label	Figure (A.4)	Yes
3.5 - A table element must contain the following elements; maximum 1 thead element, exactly 1 tbody element, maximum 10 TR elements, each TR element must contain exactly 3 TD elements, and finally the table must contain 1 caption element, or 1 summary attribute	Figure (A.5)	Yes
3.6 - All non-text content that is presented to the user has a text alternative that serves the equivalent purpose	Figure (A.6)	Partly
3.7 - Web pages have titles that describe topic or purpose.	Figure (A.7)	Yes
3.8 - Labels or instructions are provided when content requires user input.	Figure (A.8)	No

Table 6.3: Usability Experiment: Results from part 3

6.2 Web Evaluation Tool Comparison

In order to determine the quality of the Bottom-Up approach, this section will evaluate the approach and the other selected web accessibility evaluation tools presented in the introduction to this chapter, based on the challenges presented in chapter 2. This section will conclude with a comparison of the different tools.

The Ambiguity Problem is not included in the evaluation of the other web accessibility tools since it is a challenge that primarily focuses on designing requirements, and not relevant during the evaluation process of a web page.

The Generic Evaluation problem is not included in this section either (with the exception of The Bottom Up Approach), since all the web accessibility evaluation tools are able to evaluate both the Html document, and the CSS style sheet, which means that the tools provide an adequate solution for this scenario. However, it should be noted that these tools are in fact limited to only these technologies, and are not able to evaluate framework specific representations of a web page such as ASP.NET, JSP and PHP.

6.2.1 The Bottom-Up Approach

As mentioned in the introduction to this chapter, the Bottom-Up Approach has been implemented as a plug in for the WebDPF tool. The plug in consists of the WCAG 2.0 model from chapter 4 and the Bottom-Up Approach from chapter 5, and allows a user to create model discoverers that can map DOM elements in a HTML document to nodes in the WCAG 2.0 model. It can then reverse engineer any given web page and output a derived model, placed at the bottom of the meta-model hierarchy. Then by checking for conformance, any parts of the derived model that violates the meta-model will be highlighted as red, see figure 6.4a for an example.

The Transparency Problem

It was explained in chapter 2 that many of the other existing web evaluation tools generate a report containing all the invalid elements in a web page without describing how the actual evaluation process was conducted. This lack of insight into the tools inner workings, make it difficult for the domain expert to confirm that the tool is "*checking everything that it should be checking correctly*".

The Bottom-Up approach offers a solution to the transparency problem. At the end of the model understanding phase, the tool has created the derived model which is an abstract representation of the web page. The derived model functions as the lowest level specification in the modelling hierarchy, thus has a graphical representation like the rest of the WCAG 2.0 model. Moreover, the requirements that the web page needs to satisfy, are represented in the adjacent specification one level higher than the derived model, see figure 6.4b.

As covered in chapter 3, DPF is a generic graph-based meta-modelling hierarchy like approach, that specifies both the model typing and the model constraints in a graphical modelling language. Furthermore, since the WCAG 2.0 model is basically a graph consisting of nodes and edges, it is reasonably easy for the domain-expert to understand with little explanation.

The fact that the entire evaluation process boils down to a conformance check between the derived model and the WCAG 2.0 model, means that we are basically checking two things: if

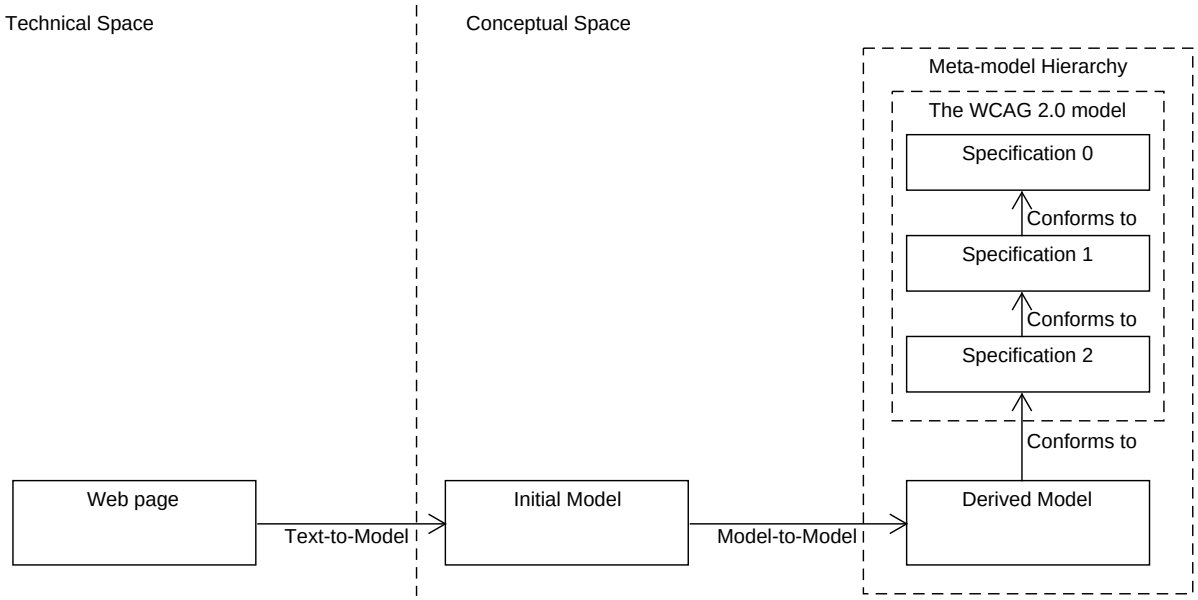


Figure 6.3: An illustration of the evaluation process included in the Bottom Up Model Validation approach

the derived model is correctly typed, and if the derived model has satisfied all of the necessary modelling constraints. All the required information is contained within the WCAG 2.0 model, which is accessible by the domain expert. It can be therefore stated that the domain expert has an adequate amount of insight into the evaluation process, which means it can be concluded that the Bottom Up approach provides a sufficient solution for the transparency problem. See figure 6.3 for an illustration.

The Barrier Problem

The barrier problem addresses the issue of handling situations where the automated web evaluation tool requires input from a person. Examples of this could be for instance determining if a header describes its corresponding section adequately.

The Bottom Up approach acknowledges that there exists requirements that are not possible to model, and therefore cannot be included in the WCAG 2.0 model. This means that there exists requirements that can not be validated automatically during the validation process. A possible fix to the Barrier Problem was presented in section 6.1, but is not included in our solution. It will however be addressed in chapter 7.

In the summary of chapter 4 this problem is addressed and the solution was employing advanced techniques in the technical space to determine for instance if an image contains text. This will of course only work for certain requirements, and there are most likely other require-

ments that are more or less impossible to automate. In conclusion, the Bottom-Up approach does not provide a solution to the Barrier Problem.

The Completeness Problem

The Completeness problem builds upon the transparency problem in that it addresses the need for the domain expert to be able to determine how many of requirements are actually included in the evaluation process, or explained another way: How complete the meta-model is. One way to determine this is making the evaluation process accessible to the domain expert.

As already addressed in the Barrier Problem, there exists requirements or parts of requirements that can not be included in an automated evaluation process since they depend on some form of human interaction. This means that the meta-model containing the requirements a web page must satisfy, should not be considered a complete model. With this is mind, it is important that the web evaluation tool ensures that domain expert is aware of what is included in the automated evaluation process, and what requires the domain expert to evaluate manually.

An adequate solution to this problem is informing the domain expert which requirements are included in the evaluation process, and how these requirements are interpreted. The WCAG 2.0 model which is accessible to the domain expert contains all the necessary information. It could be argued that the WCAG 2.0 model in its current state might be too large to be able easily understand which requirements are included, but this could be solved using a simple model-to-text transformation that generates a list of all the requirements included.

There is still however the problem of false negatives being included in the results from the evaluation process. The Bottom Up approach does provide the means to detect these. The derived model contains all the relevant html elements, and not just the ones that have failed, it could be considered a simple task for a domain-expert to examine for instance any input element they suspect to be a false negative and confirm that they are correctly labelled. Looking back at figures 5.19 and 5.20, a domain expert could easily determine if there exists any false negative input elements by confirming that there exists for each input element, either a *labelledBy*-edge going to a *label*-node, or an *attr*-edge going to a *title*-node.

The Correctness Problem

The Correctness Problem is concerned with how correct the results from the web evaluation tool are. The main goal is to reduce the number of false positives in the result, and provide the domain expert the means to determine if any results they suspect are indeed false positives.

This problem is closely related to The Completeness Problem, and therefore is solved in a similar manner. A possible solution to the Correctness Problem is providing the domain expert with a description regarding how each html element is evaluated based on its corresponding requirement. That way the domain expert is able to determine if the requirement has been in-

terpreted correctly by the developer. This information is contained within the WCAG 2.0 model.

Just as the domain expert can investigate non-flagged elements to determine if they are a false negative, the domain expert can investigate elements flagged as invalid to determine if they might be a false positive. For instance figure 5.20 would contain two input nodes that would be flagged as invalid, a domain expert could inspect the two nodes and confirm that they are invalid because of the missing edges.

The Specificity Problem

The Specificity Problem addresses challenges involving how specific the results provided by the web evaluation tool are.

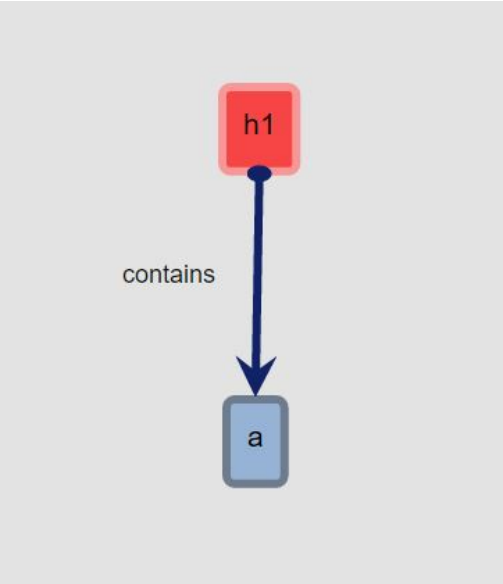
Since the evaluation process is executed in the observable conceptual space, and any inconsistencies detected during the conformance check is flagged in the corresponding parts of the derived model, the domain expert is presented with an error that reflects the violation in detail. These warnings can be presented either visually by investigating directly the flagged parts of the derived model, or textually by using a model-to-text transformation.

The Bottom-Up approach attempts to solve the specificity problem by presenting the errors identified visually using a graph-based model. The general idea is that a graphical representation of the error could explain the cause of the violation and a possible solution better than a generated textual error message. Figure 6.4 demonstrates a visual presentation of the same error illustrated in figure 6.7. A domain expert would quickly be able to discover the cause of the violation, and take the necessary measures to correct the error (i.e. pass it a long to a developer to fix).

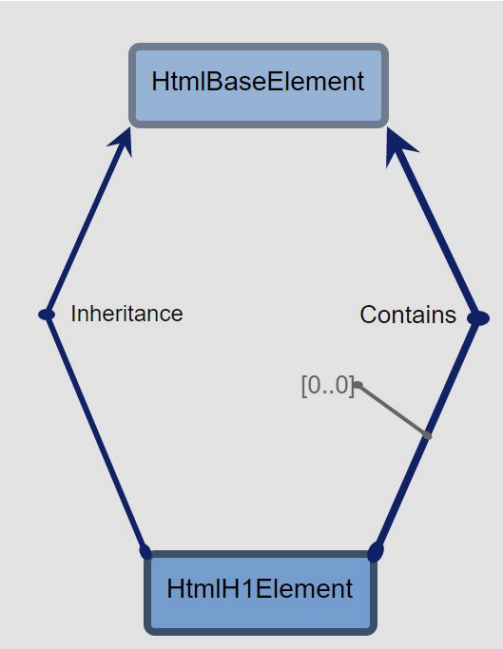
An example is the case presented in figure 6.4, a domain expert would note that the H1 element failed because it contained a Hyperlink element. Furthermore in cases where the cause is not clear, the domain expert could confer with the corresponding part of the meta-model to get a better understanding of why the html element has been flagged as a violation. It should be emphasized that all this happens in the conceptual space, and therefore does not require the domain-expert to investigate the actual html document.

The Usability Experiment in section 6.1 indicated that the domain-expert was able to pinpoint both the root cause of violation and provide a suitable correctly to the invalid element.

It should be noted however that many web accessibility evaluation tools are not limited to only presenting a textual warning, some tools such as *Wave* present its results by returning a modified version of the web page where html elements that violate the universal design requirements are highlighted.



(a) Extract from the derived model



(b) Extract from meta-model that corresponds to the extraction displayed in 6.4a

Figure 6.4: Screenshot of a warning message presented visually using a modified version of the WebDPF tool

The Generic Evaluation Problem

The Generic Evaluation Problem addresses the need for the web evaluation tool to be able to validate multiple components related to the web page. Chapter 3 defines the web page as a domain consisting of three sub-domains: *The HTML domain*, *the CSS Domain*, and *the Script Domain*. Each of these domains are defined using separate languages. There exists several criteria sets and guidelines that span across all of these domains, meaning that the web evaluation tools cannot be bound to just one of them. Furthermore, some requirements might address components that exist in the server-side code.

The Bottom-Up approach is able to overcome this problem by using platform specific model discoverers. Chapter 5 explains how the model driven reverse engineering process uses these model discoverers to extract the initial model. Every new platform the web evaluation tool needs to support, requires only that a new model discoverer is added. Since the initial model is generic, once it has been extracted, the process for validating the web page (or any other component for that matter) is executed identically regardless of what platform the component originates from. See figure 6.5 for an illustration.

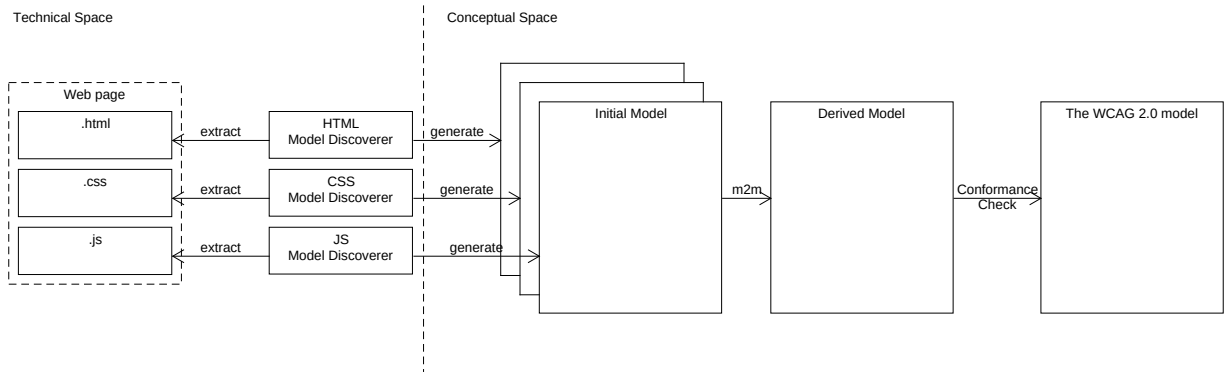


Figure 6.5: Illustration of the generic Bottom Up Model Validation Process

The Ambiguity Problem

The Ambiguity Problem focuses on complications caused by misinterpreting the requirements and misunderstandings that may occur between a domain expert and a developer. A possible solution is defining the requirements in such a way that there is little room for misunderstanding. In chapter 4 a DSML that was capable of providing a clear and precise description of the requirements was defined.

As covered in chapter 3, DSMLs are modellings languages designed for specific domains, in this case the domain in question is web page domain consisting of the CSS sub-domain and the HTML sub-domain. Since the DSML is confined to the web page domain, it's expressiveness is limited, and therefore able to describe any requirements defined using the DSML precisely. Furthermore, since the DSML is clear and concise, it may improve communication between domain experts and developers. The DSML is formalized using the WCAG 2.0 model designed in chapter 4, which encompasses the mandatory requirements and functions as the core of the web evaluation tool.

The WCAG 2.0 model attempts to provide a clear definition of the mandatory requirements the web page needs to satisfy by representing each html element included in the selected requirements as nodes, and models the actual requirements as edges between the nodes. Each edge that represents a part of a requirement has a predicate attached to it, which adds a modelling constraint to that part of the meta-model.

The WCAG 2.0 model aims at providing a description of the requirements that can be understood by both the domain expert and the developer, and therefore serve as a solution to the ambiguity problem. The results from the usability experiment discussed in section 6.1 indicates that the WCAG 2.0 model fulfils it's intended purpose, but as previously mentioned, the usability experiment is too small to provide anything conclusive.

The Modification Problem

The Modification Problem addresses two concerns. The first one being the tools ability to make alterations to the requirements in a way that allows the domain expert to confirm that the changes have been correctly implemented. The second one is providing the means for the domain-expert to make changes to the existing requirements, even though he or she does not necessarily have the knowledge or the means to makes changes to the source code.

The requirements in the Bottom-Up approach are represented in the WCAG 2.0 model which the derived model must conform to. Any desired changes made to the requirements only need to be reflected in the WCAG 2.0 model, and no where else in the web evaluation tool. This means that the domain-expert can easily review the changes made to the requirements, and confirm if they are correctly implemented. Furthermore, the plug in for the WebDPF tool, allows the user to read and edit the WCAG 2.0 model, the user can easily modify it to better model their own interpretation of the requirements. In conclusion, the Bottom-Up approach provides a solution to both concerns addressed in the modification problem.

6.2.2 AChecker

AChecker [29] is a non-commercial web-based tool for evaluating the web accessibility of a web site. Using this tool is pretty straight forward, the user provides the URL for the web page they wish to evaluate, AChecker then analyses the web page and returns a list of elements that has not satisfied the necessary requirement. The results are purely textual and consist of the html code of the failed html element, and its position in the HTML Document. The failed elements are grouped by which WCAG 2.0 requirement they have failed [1], see figure 6.7.

It's solution to the transparency problem is distributing the tool as an open source software, making the tool completely transparent to other developers. This seems like a reasonable solution, however only developers will be able to understand how the requirements are interpreted, and that's only if the code is documented well enough to communicate it's purpose to the developer. Despite having access to the source code the solution should be considered inadequate, since only developers or domain experts with the necessary programming knowledge will be able to understand its contents, and even then it might be difficult to verify that the evaluation process is correct.

AChecker attempts to solve the barrier problem by dividing all html elements that have potentially failed a requirement into three categories: *Known Problems*, *Likely Problems* and *Potential Problems*. Elements that AChecker is certain have failed a specific requirement are categorized as *known problems* [29]. For instance, a html image element that's missing the alt-attribute does most certainly not fulfil the *1.1.1 Non-text Content* requirement. In some cases it is possible to categorize the potentially failed html element as a likely problem, for instance if the alt-attribute only consists of one word, it can be assumed that the alt-attribute does not provide an adequate description of the image. The last category, *potential problems*, consists of

Checks

Priority 1

1.1 Provide a text equivalent for every non-text element.

Element	Error Type	Description	Check ID
a	Potential	Sound file must have a text transcript.	17
a	Likely	Links to multimedia require a text transcript.	20
applet	Potential	<code>applet</code> contains a text equivalent in the <code>alt</code> attribute of the <code>applet</code>.	23
area	Known	All <code>area</code> elements have an <code>alt</code> attribute.	64

Figure 6.6: Screenshot of a check list in AChecker

html elements that require a human to confirm if the html element satisfies the corresponding requirements. Even though the html image element contains an alt-attribute that consists of a sufficient number of words, it is still not possible to determine if content of the alt-attribute provides a good enough description of the image.

The trouble with AChecker's solution to the barrier problem is that the tool generates a lot of unnecessary noise, the user is only interested in which html elements that does not satisfy its corresponding requirements, and not all potential violations the web page might contain. This can also lead to problems related to *false negatives*.

In order to overcome the Completeness problem, AChecker provides a detailed list of which WCAG 2.0 requirements are included in the evaluation process. It even provides a check list of what exactly is being examined for each requirement (see figure 6.6). This provides the user with a complete and precise description of what is included in the evaluation process, which in turn makes it simpler to identify possible false negatives. The check list serves as an adequate solution to the Completeness problem. Furthermore, each item in the check list contains a detailed description of the following:

- Textual description of the requirement it belongs to
- General solution to how to repair a specific element in order for it to satisfy the requirement
- A step-by-step description of the checking procedure, including expected results and failed results.
- Examples of elements passing and failing the check.

By having access to information describing in detail how each requirement is interpreted and used to evaluate a web page makes it easier for the user to detect false positives. The explanation provided for each checking procedure, may serve as an adequate solution to the correctness problem. The only drawback here is that the details are described using a natural language, which can cause problems related to ambiguity.

Figure 6.7 demonstrates an interesting point when considering the specificity problem, which is if the root cause of the violation has been correctly identified. AChecker has successfully iden-

2.4 Navigable: Provide ways to help users navigate, find content, and determine where they are.

Success Criteria 2.4.6 Headings and Labels (AA)

Check 37: **Header nesting - header following h1 is incorrect.**

Repair: Modify the header levels so only an h1 or h2 follows h1.

Line 488, Column 6:

```
<h1 class="headline"><a class="headlinelink "  href="/har-du-rom-for-en-til.5793343-178094.html">Har ...
```

Figure 6.7: Screenshot of a warning message presented by AChecker while evaluating Ringsaker Municipality

tified a violation in the sense that a H1 element may not serve as a container for a Hyperlink element, but identifies the cause as an incorrect sequence of headers and even goes as far as to suggest modifying the header sequence so that either a H1 element or a H2 element follows the original H1 element in order to solve the problem. This suggested solution does not only fail to address the actual problem, but suggests a solution that could potentially cause further violations. A correct error message would be stating that a header element may not contain other elements, and that a possible solution would be to wrap the H1 element with the Hyperlink element instead of vice versa as it was originally implemented.

Figure 6.7 reveals also a fundamental problem with presenting the warnings textually. The warning displays only the actual line in the html document that has failed. In order to understand the context of this element (i.e. it's purpose), the domain-expert has to access the html document. This could easily prove to be problematic since firstly, the html document is not necessarily formatted neatly so that a person can easily interpret its contents. Secondly, the domain-expert might not be experienced with reading html and thus have problems understanding the context if the html element.

Finally, AChecker provides the means for the user to define their own requirements by creating their own check lists consisting of a selection of predefined checks. This means that the user is able to create their own requirements, but within the boundaries set by AChecker. Since AChecker is an open source application, it is possible for a developer to implement their own requirements from scratch. However, this means there is still the problem with confirming that the changes are implemented correctly.

6.2.3 SortSite 5 (Trial Version)

SortSite 5 is a commercial web site testing tool from PowerMapper [43], that validates a web page based on it's level of accessibility, if the web page contains any broken links, how compatible the web page is with the different common browsers and if it satisfies Html and CSS web standards. It accesses a web page by requesting it the same way a web browser would, this means that SortSite 5 obtains the web page as an html document regardless of if it is implemented with

Description	Status
Accessibility	
PDF/UA - Matterhorn Protocol 1.02	Not Checked
Section 508 of the Rehabilitation Act	Checked
Web Content Accessibility Guidelines 2.0 (WCAG 2.0)	Checked to level AA

Figure 6.8: Screenshot of what is included in the SortSite 5 evaluation process (Extract)

ASP.NET, JSP and so on [51]. It should be noted that this is the trial version of SortSite 5, the complete version might differ from what is described here.

SortSite 5 is used in a similar manner to AChecker, the user provides a URL and the tool returns a report stating which html elements have failed to pass the necessary requirements. The report is a lot more comprehensive than the one provided by AChecker and is not limited to a single web page, but actually evaluates the entire site by using web crawlers. This means that the tool is able to detect broken links. Like with the AChecker tool, the report presents all the detected errors textually. Each error included in the report is accompanied by which requirement it has failed.

Since SortSite 5 is a commercial product, it doesn't explain how it evaluates a web page or how it attempts to overcome problems related to the barrier problem or the transparency problem.

The report generated by SortSite 5 provides a superficial list of what has been checked, but not detailed enough to be considered adequate, see figure 6.8. Moreover, there is no information on how the tool interprets the included requirements, or how the web page is evaluated during the validation process. In other words, Sort Site 5 does not provide a solution to the transparency problem.

Furthermore, this tool seems to only include requirements that can be evaluated automatically, and ignore those that require some sort of human interaction. This means that the tool does not attempt to solve the barrier problem.

SortSite 5 provides a list of what is included in the evaluation process. See figure 6.8. The list is however is not detailed enough to provide any useful information. Is does not specify for instance, if all WCAG 2.0 A and AA requirements are included. The report created by the tool does provide a detailed list of all the requirements violated by the web page, but this does not really solve the completeness problem. In order to be able to determine if the results contain any false negatives, the user needs to know what requirements are included in the evaluation process, and which have been ignored.

There is also no explanation for how it interprets the requirements specified in WCAG 2.0, or how the requirements are used to evaluate the web page. This makes it difficult for the user to detect potential false positives in the results from the evaluation process. In other words, the

An image with a null ALT attribute should not have TITLE or ARIA label attributes.

If the image should be voiced use a descriptive ALT attribute. If the image is purely decorative remove any TITLE and ARIA label attributes.
Impact on users:

- JAWS : Ignores all images with ALT=".
- NVDA : Reads some images with ALT=" and a TITLE or ARIA label.
- WindowEyes : Reads images with ALT=" and a TITLE or ARIA label.
- VoiceOver : Reads images with ALT=" and a TITLE or ARIA label.

Line 331 WCAG 2.0 A F39

Figure 6.9: Screenshot of a violation detected by sortsite 5 when evaluating Bergen Municipality's web page

trial version of SortSite5 does not provide a solution for the Correctness problem.

The violations described in the report are displayed as a list and categorized based on which requirements the html element has violated. Each detected violation is well documented with a detailed description of the violation and where in the html document the violation occurred. However, the violations are described purely textual, and therefore are prone to problems related to ambiguity.

Furthermore, the report does not always provide an adequate solution to the violation. Figure 6.9 displays a screen shot from a report, the error states that an image with an *alt* attribute with the value *NULL*, should not have the *title* attribute or *ARIA* label attributes. SortSite 5 provides no suggestions for correcting the error. Should for instance the *alt* attribute be given a value, or should the *title* attribute be be removed?

The detailed report provided by SortSite 5 does solve part of the Specificity Problem in that it correctly identifies the cause of the violation, but without providing information about how the violation can be corrected, the solution should be considered as incomplete.

The trial version of SortSite 5 does not allow the user to makes changes to the requirements or add new ones. This means that SortSite 5 does not provide a solution to the Modification problem.

6.2.4 WAVE

Wave is another commercial web accessibility evaluation tool which is part of the WebAIM project. The WebAim project was started at the State University of Ohio in 1999 and focuses on improving web accessibility for individuals with disabilities. The services provided through this project have been engaged by groups such as PayPal and Ebay [59].

WAVE is used the same way as the two previous tools, the user provides a URL and the tool returns some sort of result. However, the results presented by WAVE differentiates from AChecker and SortSite 5 by being presented graphically. Figure 6.10 shows a screen shot of the tool.

Figure 6.10: Screenshot of an error reported by WAVE

WAVE provides limited information about which requirements are included in evaluation process, it is only when an element fails that the user is informed which requirement has been violated. One can only assume that being a web accessibility evaluation tool, it includes the requirements specified in WCAG 2.0. Furthermore, it does not explain how these requirements are interpreted. Thus it can be stated that WAVE does not provide a solution to the transparency problem.

The WAVE tool provides a similar solution to the barrier problem as AChecker does, by highlighting html elements that cannot be automatically evaluated. The only difference is that WAVE does not highlight these elements as warnings, but rather as additional information stating for instance that an image element has an alternative text attached to it. It is then up to the domain expert to determine if the alternative text is sufficient.

Since Wave only notifies the user about which requirements that have been violated, instead of providing a complete list of requirements included in the evaluation process, it is difficult to determine the completeness of the evaluation process. Furthermore, without knowing what is included in the evaluation process, it becomes a challenging task to identify potential false positives. In other words, Wave does not provide an satisfactory solution to the Completeness problem.

There is no information about how the requirements are interpreted or how the different html elements are evaluated based on the requirements included in the evaluation process. This means the user can not easily confirm if the tool has evaluated the html element correctly, thus detect possible false negatives. Wave does not provide an adequate solution to the Correctness problem.

As mentioned earlier, instead of listing up all the errors textually, Wave presents a modified version of the web page where every failed html element is annotated with a symbol indicating that this element has failed to pass a certain requirement. Clicking on the symbol presents the user with an error message explaining why that element has failed [59], see figure 6.10.

Since Wave presents the violations graphically, it solves the Specificity problem in a similar manner to the Bottom Up approach. The violation is presented as an annotation attached to the validated element in the web page. The annotation contains a textual description of the violation, and also provides the necessary context by indicating graphically where in the page the violation exists.

Wave does not provide any way for the user to make adjustments to the requirements or add new requirements to the evaluation process. This means that Wave does not address the Modification problem.

6.2.5 Comparison

We have selected a mix of different web accessibility evaluation tools in order to compare the bottom-up approach with both commercial and non-commercial products, and with results that are presented graphically and textually. Table 6.4 shows an overview of which challenges each tool was able to provide a sufficient solution for. As we see the Bottom-Up approach is able to solve almost all of the challenges presented in chapter 2, with the exception of the barrier problem. However as discussed in section 6.2.1, by including the *reason* notation in the WCAG 2.0 model, the bottom up approach could provide a possible solution to the barrier problem as well.

Challenge	**AChecker**	**SortSite 5 (Trial)**	**Wave**	**Bottom Up Approach**
The Transparency Problem	No	No	No	**Yes**
The Barrier Problem	**Yes**	No	**Yes**	No
The Completeness Problem	**Yes**	No	No	**Yes**
The Correctness Problem	**Yes**	No	No	**Yes**
The Specificity Problem	Partly	Partly	**Yes**	**Yes**
The Generic Evaluation Problem	**Yes**	**Yes**	**Yes**	**Yes**
The Ambiguity Problem	N/A	N/A	N/A	**Yes**
The Modification Problem	**Yes**	No	No	**Yes**

Table 6.4: Overview of which tools successfully solve challenges presented in chapter 2

Web Acessibility Evaluation Tool	**Type**	**Transparent evaluation process**	**Evaluation Results**
AChecker	Non-Commercial	partly, by viewing source code	Textual
SortSite 5 (Trial)	Commercial	No	Textual
Wave	Commercial	No	Graphical
Bottom Up Model Validation Approach	Non-Commercial	Yes, using graph-based models	Graphical

Table 6.5: A comparison of the different web accessibility evaluation tools based a certain key aspects.

Table 6.4 shows that it is actually the non-commercial web accessibility evalation tools that solves most of the challenges from chapter 2. Since Wave and SortSite 5 provide little information regarding how their products are developed, it is difficult to know if they are concerned with for instance problems related to Completeness, Correctness or Specificity. It may be that the companies that have created Wave and SortSite 5 consider themselves as experts in the field and therefore do not think it is necessary to provide the user with information about how the product evaluates web pages. This is however pure speculation.

Furthermore, of all the tools included in this section, it is only the Bottom-Up approach that provides an adequate solution to the transparency problem. AChecker attempts to solve this problem by providing access to its source code, but as mentioned in section 6.2.2 it is not considered a sufficient solution to the problem. This could indicate that the Bottom-Up approach provides an improved solution to a problem that until now has only been solved by AChecker.

Finally, table 6.5 compares some of the key aspects of the web accessibility tools included in the evaluation. It is not surprisingly, only the non-commercial products that provide some level of transparency to the evaluation process

6.3 Error Detection Comparison

In this section the selected web accessibility evaluation tools will be compared based on their ability to detect and correctly identify each violation contained within a web page. First each tool will evaluate a test web page containing elements that satisfy and violate several of the selected universal design requirements. The results will be presented in table 6.6. After the test page has been evaluated, each tool will evaluate a selection of Norwegian municipalities. A summary of the results will be displayed in table 6.7. Note that the results presented in table 6.7, only include the violations of the requirements included in the WCAG 2.0 model, and not the complete results reported by each tool.

Table 6.6 shows that the Bottom Up approach was able to detect roughly the same amount of violations in the test page as the other tools, however it only registered half of errors im-

Requirement	AChecker	SortSite 5 (Trial)	Wave	Bottom Up Approach	# of errors
1.1.1 Non-text Content	1	1	1	1	1
1.4.5 Images of Text	0	0	0	0	1
1.4.9 Images of Text (No Exception)	0	0	0	0	1
1.3.1 Info and Relationships	2	1	2	1	2
2.4.6 Headings and Labels	1	0	1	1	1
1.4.8 Visual Presentation	0	0	0	0	0
2.4.10 Section Headings	0	0	0	0	0
1.4.3 Contrast (Minimum)	0	0	0	0	0
1.4.6 Contrast (Enhanced)	0	0	0	0	0
Total errors found	4	2	4	3	6

Table 6.6: Overview of how many errors each tool found in the test page and actual number of errors implemented

Municipality	AChecker	SortSite 5 (Trial)	Wave	Bottom Up Approach
Bergen Municipality [5]	30	14	5	7
Os Municipality [38]	1	14	4	3
Ringsaker Municipality [46]	6	7	5	7
Oslo Municipality [39]	1	1	1	5
Ålesund Municipality [3]	18	7	10	1

Table 6.7: A summary of how many errors each tool found at each municipalities web page

plemented into the web page. Furthermore, the table shows that none of the tools have the capability to determine if an image contains any text or not.

Table 6.7 shows an overview of how many violations each web accessibility evaluation tool found in each of the municipal web pages selected for the evaluation. The overview only contains the number of actual errors, and not potential violations and warnings. The first thing that should be addressed is the variation in how many errors each tool found, AChecker registers 30 violations, but Wave only detects 5. With the exception of Oslo Municipality, it seems that none of the tools are able to reach a consensus on how many WCAG 2.0 have actually been violated. This could indicate that the different web accessibility evaluation tools have interpreted the requirements differently, causing violations detected by one tool to be registered as a warning or ignored all together by the other tool. The lack of consensus between the tools, supports one of the fundamental question from chapter 2: *What is actually being evaluated and is it being evaluated correctly?* See tables in appendix C for a more detailed overview of the errors detected in each municipal web page.

Since there is no consensus between the other web accessibility tools, it is difficult to determine the correct number of violations each municipal web page contains, and therefore difficult

to determine how well the Bottom Up approach did in a real world scenario. However, with derived model at the users disposal, it was easy to determine that the detected violations were legitimate. It is also possible to conclude that the results reported by the Bottom Up approach coincides roughly with what the other tools reported (with the exception of Oslo Municipality).

6.4 Summary

The Evaluation of the Bottom Up approach, has shown some promising results with regards to not only the challenges described in chapter 2, but also based on the results from the usability experiment discussed in section 6.1 and when compared with other existing state-of-the-art web accessibility evaluation tool in sections 6.2 and 6.2.5.

The WCAG 2.0 model formalized in chapter 4 contains a clear and concise definition of the requirements included in the web evaluation tool. In doing so it provides a solution to the ambiguity problem.

Furthermore, the WCAG 2.0 model contains all the necessary information for the domain expert to detect possible false negatives and false positives that might exist in the results from the evaluation process. Thus, the WCAG 2.0 model also provides a possible solution for the completeness problem and the correctness problem.

The Bottom Up approach is able to provide an adequate level of transparency into the evaluation process, by evaluating the web page in the conceptual space. None of the other tools discussed in this chapter are able to do this. The entire evaluation process is essentially just a conformance check between an abstract model representation of the web page, and the WCAG 2.0 model. Because of this, problems such as the modification problem are easily overcome, since any changes made to the requirements are confined to the WCAG 2.0 model, and therefore included naturally in the evaluation process.

The only challenge from chapter 2 the Bottom Up approach does not provide a solution for is the barrier problem. However, the usability experiment provided a reasonable solution to this problem. By defining a new notation, it is possible to inform both the domain expert and the evaluation process that a specific edge requires the attention from a user during the evaluation process.

However it should be noted that the Bottom Up approach does require some explanation before use, meaning that this tool might be the least user friendly of all the web accessibility evaluation tools discussed in this chapter.

Even though the results from usability experiment can not be used to conclude anything, they do give an indication that the WCAG 2.0 model may be a suitable way of communicating the mandatory requirements between a developer and domain expert. The usability experiment also showed that the graph-based result from evaluation process is able to describe detected violations efficiently.

Finally, the results from the comparison between the state-of-the-art web accessibility evaluation tools and the Bottom Up approach have yielded two interesting points: Firstly, there is a lack of consensus between the different tools in regards to how many accessibility violations each municipal web page contains. This could indicate that the WCAG 2.0 standard lacks a certain level of formalization, causing multiple interpretations of the requirements. Secondly despite the lack consensus, the Bottom Up approach was able to detect roughly the same amount of violations as the other tools.

Chapter 7

Conclusion

As more human-based services are converted to digital web-based services, it is important to ensure that not only the quality of the service is acceptable, but that it is accessible to all users regardless of any disabilities they may have. In order to ensure this, requirements, recommendations and guidelines have been established by interested parties such as DIFI. It is however, a time consuming and tedious process to manually examine existing web pages to confirm that they fulfil the necessary requirements. The purpose of this thesis has been to provide a tool that could automate the web evaluation process using methods and techniques from MDE.

In this thesis there has been presented two alternative approaches to how meta-modelling can be used to evaluate an existing web page. The result was a web evaluation tool that not only could detect violations contained within a web page using MDE, but it also provided solutions to several problems that existing web evaluation tools are exposed to. The most noteworthy one being the lack of transparency in the evaluation process, where the Bottom-Up approach was successfully able to provide insight into the evaluation process which can be interpreted by domain experts and developers.

In chapter 2 several challenges were presented, most of them originated from other papers that addressed concerns related to existing web evaluation tools, the rest were identified as potential problems related to the cooperation between a domain expert and a developer when adding new requirements to the web evaluation tool or modifying existing ones. In chapter 6, the solution has shown that is was able to overcome 7 of the 8 challenges: *the transparent problem*, *the completeness problem*, *the correctness problem*, *the specificity problem*, *the generic evaluation problem*, *the ambiguity problem* and *the modification problem*.

An interesting case was discovered in chapter 6 while comparing the solution's error detection capabilities with other existing web accessibility evaluation tools, there was a lack of consensus in the number of violations detected. The fact that the tools reported different numbers of web accessibility violations for each page, could indicate that despite each WCAG 2.0 requirement being thoroughly described, the requirements are still ambiguous, and therefore prone to being misinterpreted. This concern was addressed in the ambiguity problem described in

chapter 2, and in chapter 4 a DSML was designed to provide a precise definition of the requirements, then formalized as the WCAG 2.0 model in an attempt to overcome this challenge. The usability experiment, despite being to small to provide anything conclusive, indicated that the requirements included in the WCAG 2.0 model were easy to interpret, and left little room for misunderstanding.

In conclusion, the proposed solution which is comprised of the WCAG 2.0 model and the Bottom-Up approach, has proven to be a viable automated web evaluation tool that addresses several critical concerns. Based on how many of the challenges from chapter 2 the solution overcame, and it's abililty to detect violations in a web page, it can be declared that the web evaluation tool has successfully fulfilled it's intended purpose. Furthermore, the work conducted throughout this thesis has introduced a new possible application for MDE.

The solution is actually very flexible. Any changes to the requirements can easily be reflected in the WCAG 2.0, and thereby included in the evaluation process. By adding new platform specific model discoverers, it is not necessary to limit bottom-up approach to evaluating static web pages, it is entirely possible to evaluate other parts of a web based system, such as the security or other server-side components.

The solution has made it possible to formalize a set of requirements as a meta-model, then use that meta-model evaluate real world artefacts. In theory the solution can use a meta-model to evaluate any piece of accessible data. This could be considered as an innovative technique for reducing the gap between the conceptual world of modelling and the real world.

The solution does however contain several limitations that will be discussed in section 7.1. Moreover, the fact that the solution is a prototype means that there is still some further work that must be done before it can be considered complete, this will be explained in section 7.2.

7.1 Current Limitations

Despite the success shown in chapter 6, the solution described in this thesis has several limitations. In chapter 2, the barrier problem was introduced as a challenge that addressed the complications involved when evaluating elements in a web page that require human interaction. The solution does not provide an answer to this problem, and even though the usability experiment in chapter 6 did yield a potential fix, it still does not automate the entire evaluation process as originally desired. However, considering the problems discussed in chapter 1, it might be sufficient with a semi-automated evaluation process, in which case the barrier problem could be considered solved. There are ways these types of checks can be fully-automated, but that would require advanced techniques such as image processing, something that is executed outside of the scope of the WCAG 2.0 model and the Bottom-Up approach.

Another limitation with the solution is the scope of the evaluation process. As described in chapter 5, the Bottom-Up approach treats the web page as a static document and therefore any

user interaction is ignored. This means any changes to the web page that may occur as a result of user interaction will not necessarily be included in the evaluation process. In other words, only the web pages initial state is evaluated.

7.2 Further Work

As mentioned in the introduction to this chapter, the solution described in this paper is a prototype and therefore requires some further work before being able to used to evaluate actual web pages.

7.2.1 Completing the WCAG 2.0 Model

Only a handful of requirements from the WCAG 2.0 standard were included in the WCAG 2.0 model. In order for the proposed solution to be considered a fully fledged web evaluation tool, the WCAG 2.0 model needs to be completed. This involves including the 35 A and AA WCAG 2.0 requirements in the WCAG 2.0 model. Furthermore, as mentioned in the introduction to chapter 4, the WCAG 2.0 model only includes elements and attributes supported by HTML5, ideally the web evaluation tool should also support HTML 4.01 and XHTML 1.0. It might also be beneficial to create a separate specification that models the CSS standard.

Universal Design is just one of several standards mentioned in chapter 1, by creating separate models for each standard, the Bottom-Up approach can easily evaluate a web page based on a model representation of a specific standard.

7.2.2 Conducting a Larger Usability Experiment

In chapter 6 the results from a conducted usability experiment is discussed. Since the experiment consisted of only two participants, it is not possible to conclude that a graph-based model such as the WCAG 2.0 model, can substitute requirements specified in a natural language. A larger usability experiment consisting of many more participants should be conducted in order to determine if the WCAG 2.0 can in fact serve as a clear and concise way of communicating the WCAG 2.0 requirements between domain experts and developers.

7.2.3 Presenting Models Textually

Presenting the violations using a graph-based model might not always be desirable, there may be scenarios where the user needs a simple overview of which elements in a web page has failed, or the user might want a list of which requirements are included in the evaluation process.

Graph-based models are useful in situations where specific artefacts such as invalid web page elements need inspecting, or certain interpretations of the included requirements need to be checked, but might be ill-suited when the user needs a comprehensive overview. With this in mind, it might be beneficial to implement a model-to-text transformation that could generate for example an overview of what is included in the evaluation process in a natural language.

This is not exactly a complex problem to overcome, but was not included in this thesis due to the lack of time.

7.2.4 Model Verification

As mentioned in chapter 3, model conformance is just the tip of the iceberg when discussing model validation and verification, another factor that should be examined is the model's consistency. This is addressed in the satisfiability problem which investigates if a meta-model contains any contradicting constraints, if so it will be impossible for a model to successfully conform to the meta-model. This problem could be applied to the WCAG 2.0 model, where it could be determined if a web page can actually satisfy all of the requirements included in the WCAG 2.0 model, or that changes to the requirements in WCAG 2.0 model does not conflict with other existing requirements.

7.2.5 Model Adequacy

The Bottom-Up approach does not evaluate the actual web page, but rather an abstract model representation of the web page. This means that in order for the evaluation process to present accurate results, the model representation needs to be adequate. An adequate model is a model that contains all necessary information that is relevant to the evaluation process. In order to ensure that the model is adequate, some sort of automated mechanism should be included in solution. This is addressed in the adequacy problem mentioned in chapter 5.

Appendix A

Acronyms

ADM Architect-driven Modernization

ADMTF Architecture-Driven Modernization Task Force

CORBA Common Object Request Broker Architecture

CSS Cascade Styling Sheet

DIFI The Agency for Public Management and eGovernment

DPF Diagram Predicate Framework

MDA Model Driven Architecture

MDE Model Driven Engineering

MDRE Model Driven Reverse Engineering

MDSE Model Driven Software Engineering

MOF Meta Object Facility

M2M Model-to-Model transformation

M2T Model-to-Text transformation

OCL Object Constraint Language

OMG Object Management Group

RE Reverse Engineering

TBRE Template Based Reverse Engineering

T2M Text-to-Model transformation

UML Unified Modelling Language

XMI XML Metadata Interchange

Appendix B

DPF Specifications

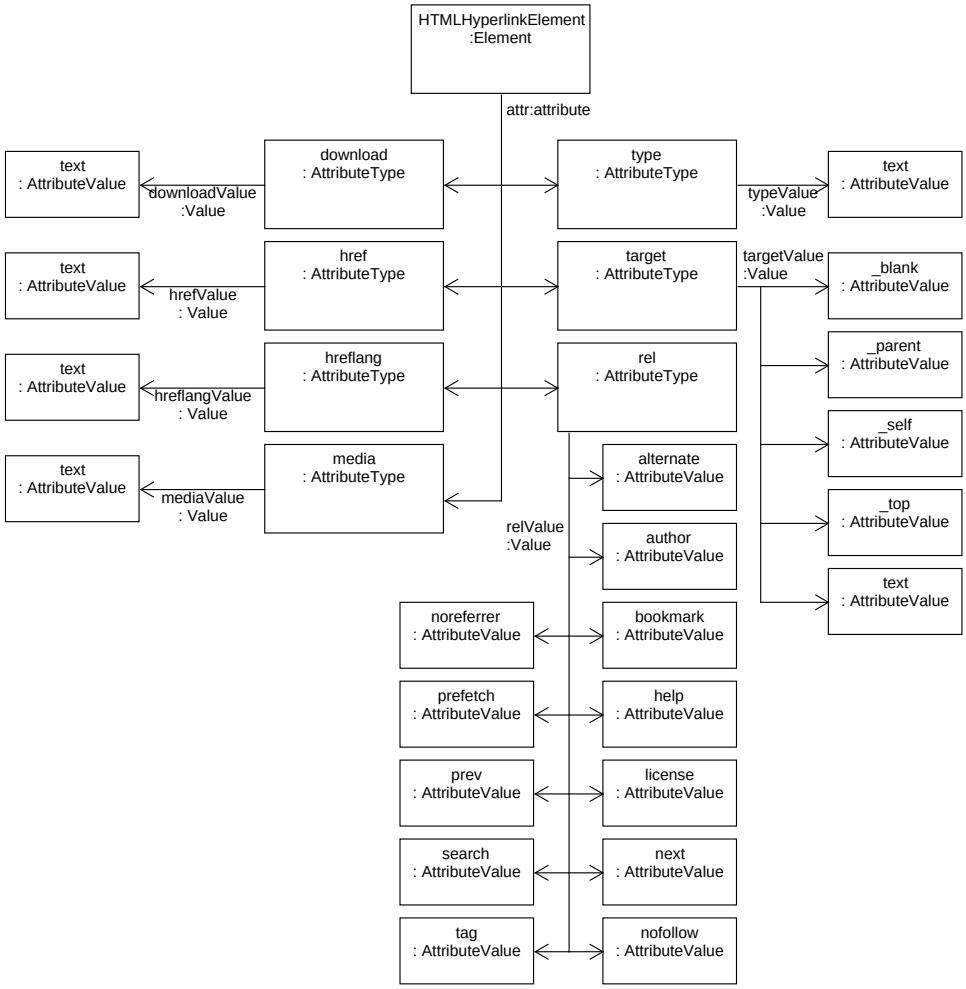


Figure B.1: The Hyperlink Element Specification

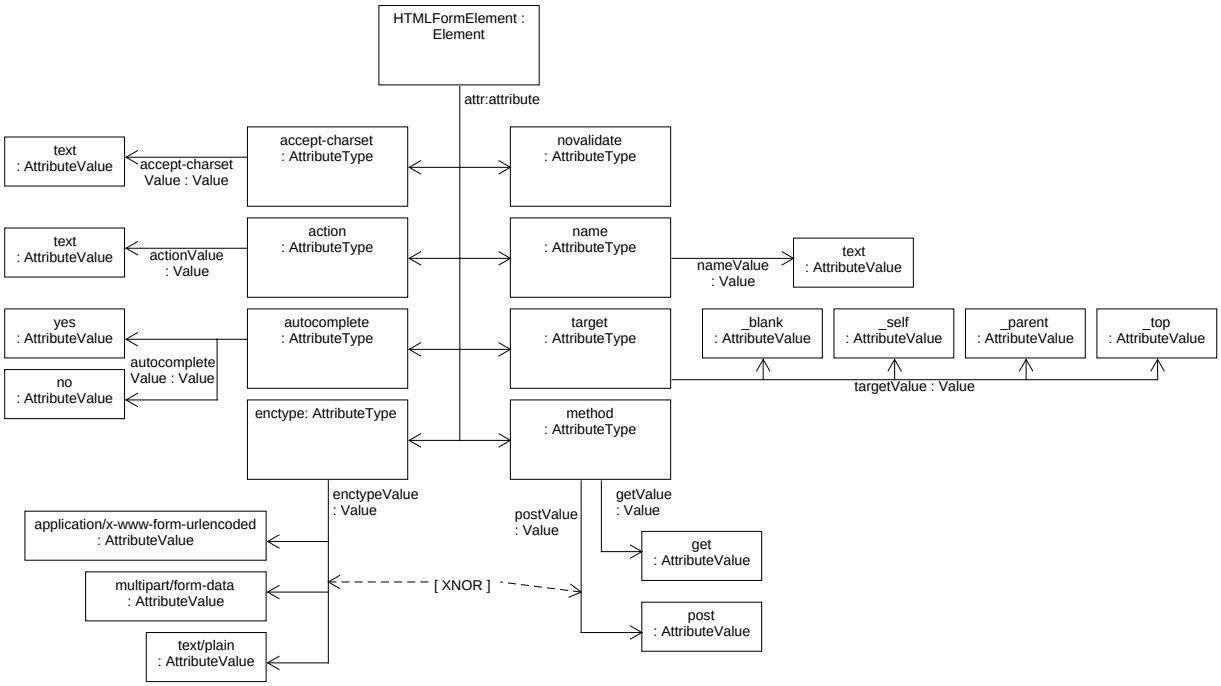


Figure B.2: The Form Element Specification

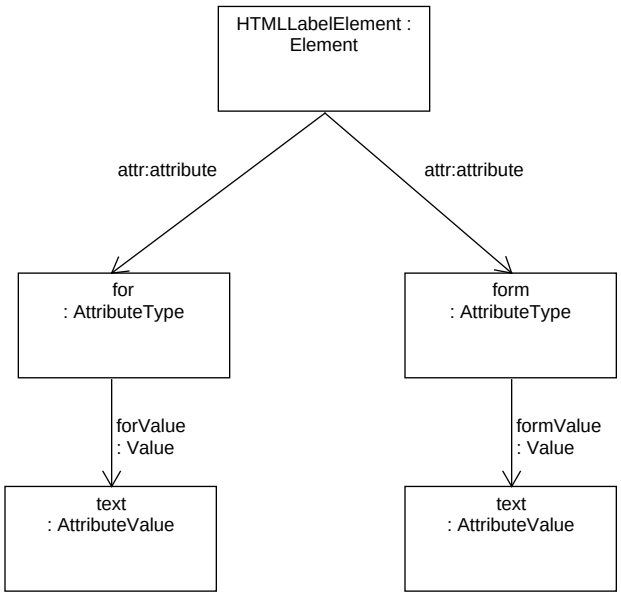


Figure B.3: The Label Element Specification

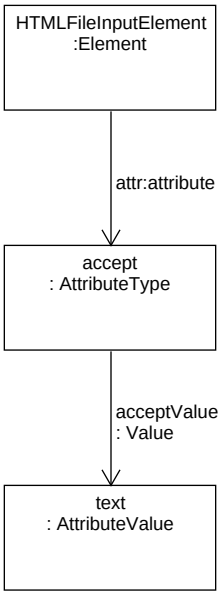


Figure B.4: The File Input Element Specification

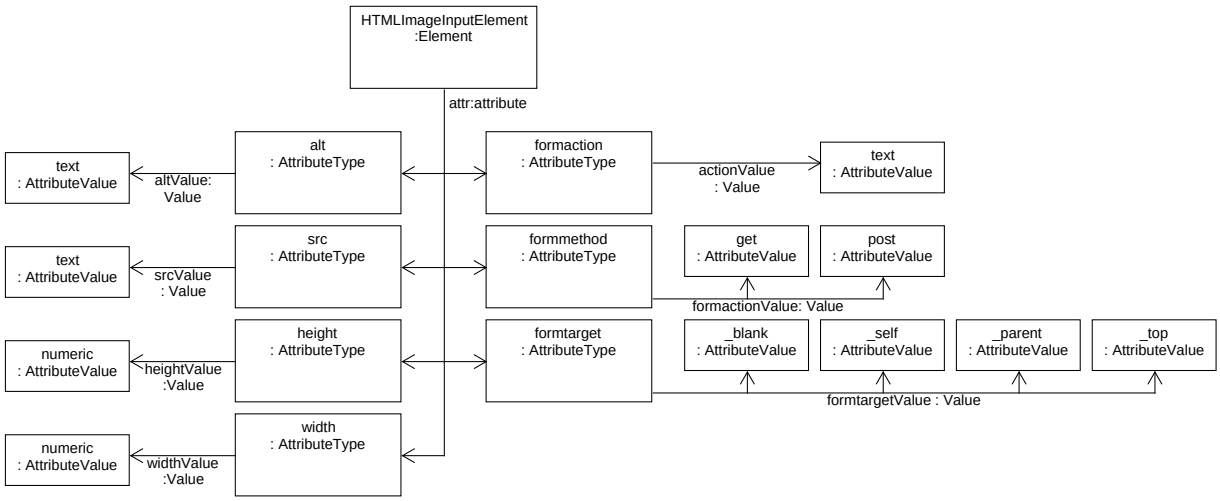


Figure B.5: The Image Input Element Specification

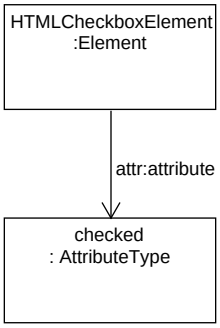


Figure B.6: The Checkbox Element Specification

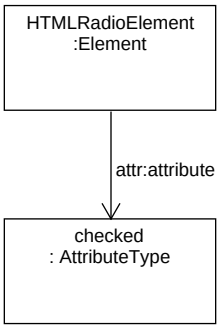


Figure B.7: The Radio Element Specification

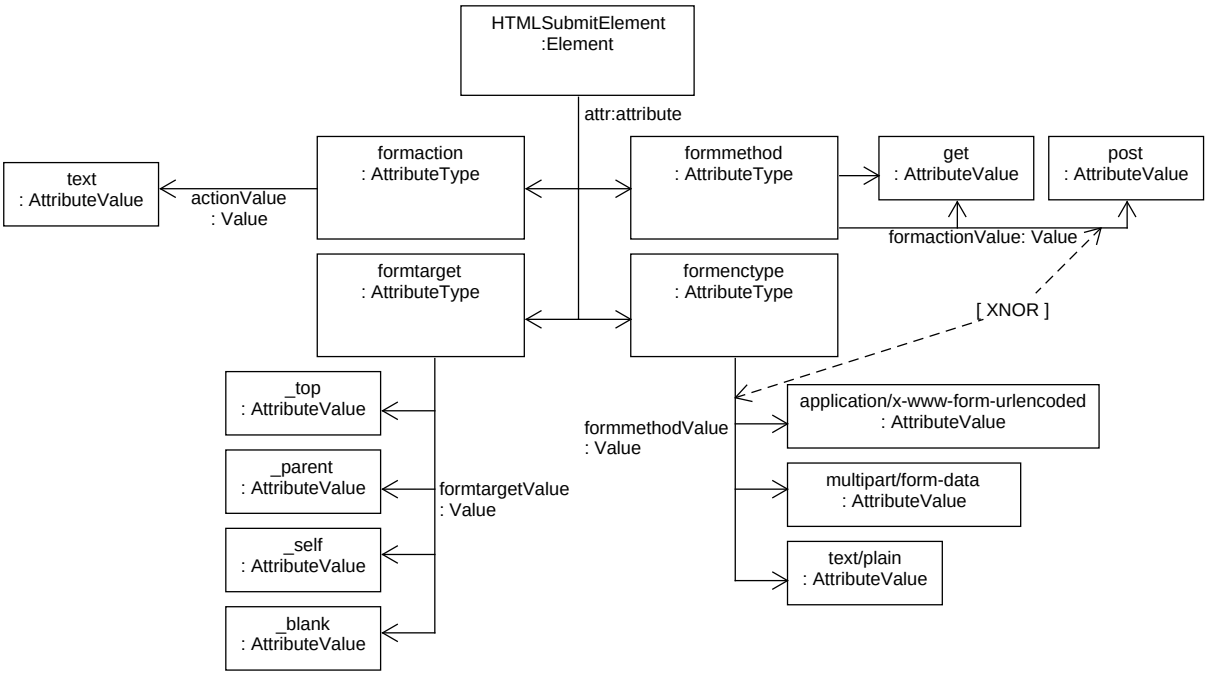


Figure B.8: The Submit Element Specification

Appendix C

Error Detection Overview per Municipality

Requirement	**AChecker**	**SortSite 5 (Trial)**	**Wave**	**Bottom Up Approach**
1.1.1 Non-text Content	18	7	4	0
1.4.5 Images of Text	0	0	0	
1.4.9 Images of Text (No Exception)	0	0	0	
1.3.1 Info and Relationships	2	0	0	
2.4.6 Headings and Labels	8	7	8	7
1.4.8 Visual Presentation	0	0	0	0
2.4.10 Section Headings	0	0	0	0
1.4.3 Contrast (Minimum)	2	0	0	0
1.4.6 Contrast (Enhanced)	0	0	0	0
Total errors found	30	14	5	7

Table C.1: Overview of how many errors each tool found at https://www.bergen.kommune.no/

Requirement	**AChecker**	**SortSite 5 (Trial)**	**Wave**	**Bottom Up Approach**
1.1.1 Non-text Content	0	5	4	0
1.4.5 Images of Text	0	0	0	
1.4.9 Images of Text (No Exception)	0	0	0	
1.3.1 Info and Relationships	0	0	0	
2.4.6 Headings and Labels	1	2	0	3
1.4.8 Visual Presentation	0	0	0	0
2.4.10 Section Headings	0	0	0	0
1.4.3 Contrast (Minimum)	0	7	0	0
1.4.6 Contrast (Enhanced)	0	0	0	0
Total errors found	1	14	4	3

Table C.2: Overview of how many errors each tool found at https://oskommune.no/

Requirement	**AChecker**	**SortSite 5 (Trial)**	**Wave**	**Bottom Up Approach**
1.1.1 Non-text Content	0	0	0	0
1.4.5 Images of Text	0	0	0	
1.4.9 Images of Text (No Exception)	0	0	0	
1.3.1 Info and Relationships	5	0	4	7
2.4.6 Headings and Labels	1	2	1	0
1.4.8 Visual Presentation	0	0	0	0
2.4.10 Section Headings	0	0	0	0
1.4.3 Contrast (Minimum)	0	5	0	0
1.4.6 Contrast (Enhanced)	0	0	0	0
Total errors found	6	7	5	7

Table C.3: Overview of how many errors each tool found at http://www.ringsaker.kommune.no/

Requirement	**AChecker**	**SortSite 5 (Trial)**	**Wave**	**Bottom Up Approach**
1.1.1 Non-text Content	0	0	0	0
1.4.5 Images of Text	0	0	0	0
1.4.9 Images of Text (No Exception)	0	0	0	0
1.3.1 Info and Relationships	0	0	0	4
2.4.6 Headings and Labels	1	1	1	1
1.4.8 Visual Presentation	0	0	0	0
2.4.10 Section Headings	0	0	0	0
1.4.3 Contrast (Minimum)	0	0	0	0
1.4.6 Contrast (Enhanced)	0	0	0	0
Total errors found	1	1	1	5

Table C.4: Overview of how many errors each tool found at https://www.oslo.kommune.no/

Requirement	**AChecker**	**SortSite 5 (Trial)**	**Wave**	**Bottom Up Approach**
1.1.1 Non-text Content	7	6	4	0
1.4.5 Images of Text	0	0	0	0
1.4.9 Images of Text (No Exception)	0	0	0	0
1.3.1 Info and Relationships	10	0	1	
2.4.6 Headings and Labels	1	0	1	0
1.4.8 Visual Presentation	0	0	0	0
2.4.10 Section Headings	0	0	0	0
1.4.3 Contrast (Minimum)	0	1	0	0
1.4.6 Contrast (Enhanced)	0	0	0	0
Total errors found	18	7	10	1

Table C.5: Overview of how many errors each tool found in http://www.alesund.kommune.no/

Usability Experiment

Thomas Calder

June 2016

The Department of Computer Engineering
Bergen University College
&
The Department of Informatic
University of Bergen

Appendix D

Introduction

The purpose of this document is to present the results from a conducted usability experiment in order to determine if a specific model defined using the *Diagram Predicate Framework (DPF)* can be used to provide a common understanding for individuals with different areas of expertise within a certain domain. This document is to serve as a supplement to the corresponding master thesis.

This paper proposes the following situation: A consultant from a Norwegian software company is sent to attend a course arranged by *The Agency for Public Management and eGovernment (DIFI)* in order to obtain a better understanding of the newly established mandatory criteria set for *Universal Design*. Upon fulfilling the course, the consultant has the necessary knowledge to determine if a web page satisfies all of the criteria specified in the Universal Design criteria set, and is therefore put in charge of modernizing the software companies product in order to satisfy the criteria. However the consultant does not have the expertise to make these changes, and a software developer is assigned to the project. The software developer is not familiar with the Universal Design requirements, and thus is dependent on the consultant to relay the information the developer needs to modernize the product. The question is how can the consultant describe these requirements to the software developer in such a way that there is little room for misunderstanding. The solution is defining the requirements using a domain specific modelling language (DSML), then using the DSML to formalize a model that both the consultant and developer can understand.

This paper will present the results of a small usability experiment consisting of a developer and a consultant from the Norwegian Software Company Acos AS. Both participants have been involved in earlier projects involving Universal Design, and are therefore familiar with the domain. This experiment should be considered as an exploratory investigation to determine if it is possible to use the DPF model as a means for communicating web page requirements between individuals with different areas of expertise. Due to the size of the experiment, it is not possible to generalize the results of this experiment. A much larger experiment consisting of more participants needs to be conducted in order to determine if the DPF model does provides a common understanding for the individuals. See figure D.1 and figure D.2 for a description of

the DPF model.

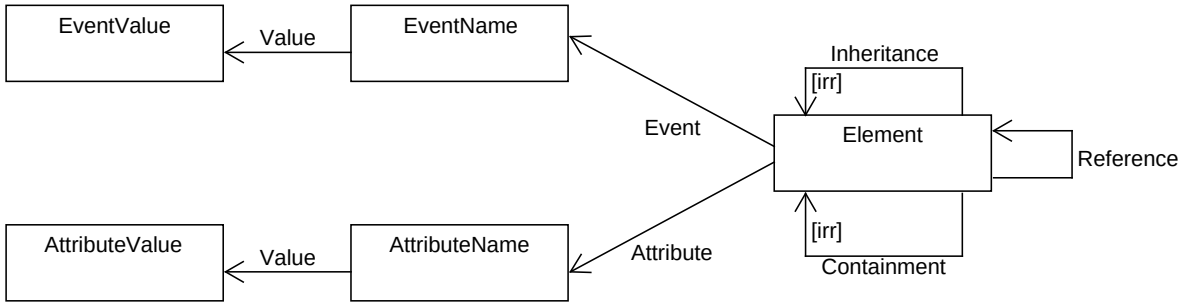


Figure D.1: DPF model: Specification $\mathfrak{S}_1$

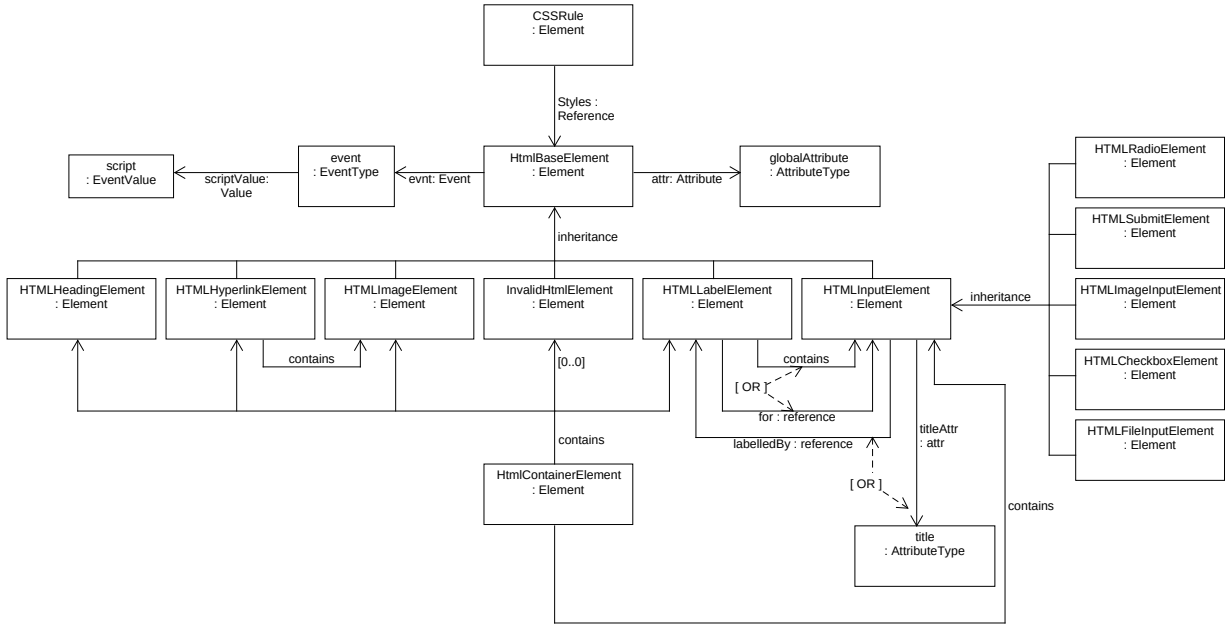


Figure D.2: DPF model: Specification $\mathfrak{S}_2$

This paper will consist of the following chapters:

- Chapter E describes the results from the usability experiment
- Chapter F consists of a short discussion concerning the results and a conclusion.

Appendix E

The User Experiment

In order to determine if the developer and the consultant were both able to understand the DPF model, a series of assignments were designed for them to solve together as a group. The results from part one and part two were obtained via observations by the author of this paper. Results from part three were obtained by asking the participants to solve the tasks on a piece of paper which could be included in the results. The author provided no help to the participants during the usability experiment with the exception of assignment 1.4 where some clarification was needed.

The experiment consisted of three parts. The first part involved showing the participants a set of DPF models, and asking them to describe the requirement that was included in the model. Part two focused on model conformance. The participants were asked to identify why parts of the model did not conform to the meta model. Part three the participants were given a series of requirements they had to model. Before the usability experiment started, the participants were given a short explanation of how the model should be understood.

E.1 Part 1: Interpreting Example Model

In part 1 the participants were shown nine different DPF models and for each model asked to explain the requirement and give an example of a situation where the requirement is satisfied or an example where it is violated. This was in order to determine if the both participants were able to understand the models.

E.1.1 Assignment 1.1

The participants were shown figure A.1, and described the requirement as the following: *Every image element needs to contain exactly one alt attribute.*. The requirement is satisfied if an image

element contains an alt attribute, and is violated if it does not have one.

E.1.2 Assignment 1.2

The participants were shown figure A.2, and after some discussion of what an invalid element is, described the requirement as the following: *No DIV element may contain a Html element that is incorrectly implemented.* The requirement is satisfied if the DIV does not contain any invalid Html elements, and is violated if it does.

E.1.3 Assignment 1.3

The participants were shown figure A.3. There was some confusion about combining the OR-constraint with the multiplicity constraints. The consultant felt that it was unnecessary to have both since the OR-constraint implied that the Input element can have 0 or 1 title attributes and 0 or 1 label elements. The requirement was described as: *An Input element must either have the title-attribute, be labelled by a label element or both.* An Input element that lacks both violates the requirement.

E.1.4 Assignment 1.4

The participants were shown figure A.4. There was some uncertainty about the *exists*-reference since the developer pointed out that header elements in a Html document, do not contain references to each other. After some explanation from the author, the participants had a better understanding of the model and provided the following definition: *Every header must appear sequentially, a H3 element may not be implemented without a H2 element appearing before it, and a H2 element may not be implemented with a H1 element appearing before. The H1 element is not dependent on any other header.* With the exception of the H1 header, any header that is implemented out of sequence will violate the requirement.

E.1.5 Assignment 1.5

The participants were shown figure A.5 and provided the following requirement: *A label must either contain an input element or refer to it using the for attribute, or both.* A label that is missing the for-attribute and does not contain an input element violates the requirement.

E.1.6 Assignment 1.6

The participants were shown figure A.6, and there was a short discussion among them of what the model was describing. One of the points they discussed was how the Contrast-constraint operated since it was not like the logical constraints they had seen earlier. There was also some discussion if the foreground colour was being compared to the labels background colour, or if it was the web pages backgrounds colour. They concluded with the following definition: *Every the background colour and the foreground colour of every label element must have a contrast of 4.5:1*.

E.1.7 Assignment 1.7

The participants were shown figure A.7. This model required a bit of time to study since it consisted of more nodes and edges than the previous ones. The participants provided the following explanation: *A form element must contain at least one label element and one input element. Each label element must contain exactly one input element. Each input element must contain either a title attribute, or be labelled by a label element.*

E.1.8 Assignment 1.8

The participants were shown figure A.8, and provided the following explanation: *A hyperlink element may contain a H1-element, but not vice versa.* A H1 element violates the requirement if it contains a hyperlink element.

E.1.9 Assignment 1.9

The participants were shown figure A.9, and provided the following explanation: *The Html element must contain the lang-attribute.* The Html element will be considered invalid if it is missing the lang-attribute.

E.2 Part 2: Understanding Model Conformance

In part 2 the participants were shown four figures consisting of a model conforming to it's meta-model. Each model fails to conform to some part of the meta-model. The participants were asked to identify why parts of the meta-model failed, and what changes had to be made to the model in order for it to conform to the meta-model. Each invalid node in the figures in this section are highlighted in red.

E.2.1 Assignment 2.1

The participants were shown figure A.1, and were able to identify the problem as the following: the IMG element is missing the alt-attribute. *In order for the model to conform to the meta-model, the IMG element needs to refer to an ALT attribute using the attr edge.*

E.2.2 Assignment 2.2

The participants were shown figure A.2, and were able to determine that a header element may not contain a HyperLink element. In order for the model to conform, the *h1* element must not contain the *a* element.

E.2.3 Assignment 2.3

The participants were shown figure A.3. Despite the model being more complex than the previous ones, they were able to quickly determine for each failed element why it was incorrect and how to correct it. The participants were able to determine that each TR element must contain at least on TD element, the failed TR element needs to refer to at least one TD element in order to be valid. The *theader* was invalid because it contained a TD element, and not a TH element. The theader element could be fixed by changing the *headercelle* from a TD to a TH. Finally in order for the Table element to be valid, a caption element needs to be added, or the summary attribute.

E.2.4 Assignment 2.4

The participants were shown figure A.4, and were easily able to identify the problem as an error in the sequence of header elements. The suggested fix here was changing the *H2* element to a *H1* element, the *H3* elements to *H2* elements, and the *H5* element to a *H3* element.

E.3 Part 3: Designing new Requirements

In part 3 the participants were given requirements from the observer that they had to model. It should be noted that several of these requirements are not taken from the Universal Design criteria, but are made up by the observer in order to create some diversity in the requirements.

E.3.1 Assignment 3.1

The first requirement to be modelled was: *A tablecell may not contain any input elements.* The participants created the following model shown in figure A.1.

E.3.2 Assignment 3.2

The second requirement given to the participants was: *A label may not contain more than input element.* The participants created the following model shown in figure A.2.

E.3.3 Assignment 3.3

The third requirement was: *A label element may not contain an input element, and refer to an input element.* The participants created the following model shown in figure A.3.

E.3.4 Assignment 3.4

The forth requirement that the participants had to model was: *An Input element must be referred to by a label* The participants created the following model shown in figure A.4.

E.3.5 Assignment 3.5

The fifth requirement was a rather large and complex requirement consisting of the following: A table element must contain the following elements; max 1 thead element, exactly 1 tbody element, max 10 TR elements, each TR element must contain exactly 3 TD elements, and finally the table must contain 1 caption element, or 1 summary attribute. The participants created the following model shown in figure A.5.

E.3.6 Assignment 3.6

The remaining requirements in part 3 are taken straight out of the Universal Design criteria. The first Universal Design criteria to be modelled was: *All non-text content that is presented to the user has a text alternative that serves the equivalent purpose* The participants created the following model shown in figure A.6. Here it should be noted that the participants realized that it was necessary for a human to determine it the text alternative served the equivalent purpose, and therefore added a new constraint called *reason* to indicate that a human needs to determine if this requirement is satisfied or not.

E.3.7 Assignment 3.7

The second Universal Design requirement to be modelled was: *Web pages have titles that describe topic or purpose.* The participants created the following model shown in figure A.7.

E.3.8 Assignment 3.9

The third and final Universal Design requirement to be modelled was: *Labels or instructions are provided when content requires user input.* The participants created the following model shown in figure A.8.

Appendix F

Conclusion

The results of the usability experiment seem promising. Both the developer and the consultant were able to quickly understand the contents of each model shown to them, as well a design their own models.

There were a few interesting points worth mentioning. The first one being how the participants chose to model requirements that required input from a human, i.e. determine if a alternative text describes its corresponding image adequately. The DPF model acknowledges that these types of requirements exists, but chooses to not to include them in the model since it is not possible to automate such validation with DPF. The participants were also aware of this limitation, but decided to implement the edge *reason*, which indicated that a human needed to confirm if that part of the model is valid.

The second interesting point was during part 2 where the participants were asked to correct the models that had failed to conform to their corresponding meta-models. The participants commented that even though they would of preferred to have access to the actual web page during validation, the model made it simpler to not only identify which elements were invalid, but also determine what the root cause that prevented the invalid element from conforming to the meta-model. For instance figure A.4 shows two invalid headers, a typical error message could indicate that the h1 and h4 element are missing, and by adding them to the web page all the header elements will be valid.

However, when the participants viewed figure A.4, they determined the cause of the error was not missing headers, but rather that the headers were implemented incorrectly. The headers would successfully conform to the meta-model by changing the existing headers so that they appear sequentially, instead of adding new ones.

The interesting point here is that by presenting the errors graphically using the DPF model, the participants were able to pinpoint the root of the error and provide a more correct solution.

In conclusion this paper can state that the DPF model was able to create a common un-

derstanding of the requirements it attempted to model for the two participants. Furthermore, the experiment showed how quickly these individuals with no prior knowledge to the Diagram Predicate Framework were able to understand the the DPF model, and use it to create their own models. However, the results of this experiment can not be generalized to a larger population due to its size.

Appendix A

Models from Part 1

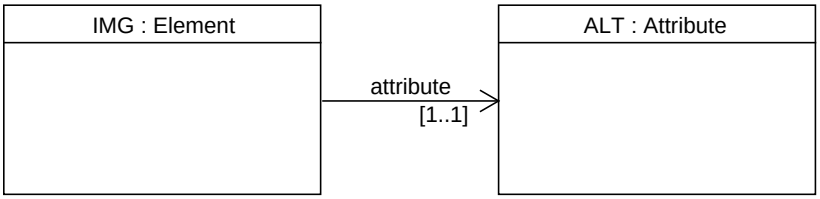


Figure A.1: Usability Experiment: Assignment 1.1

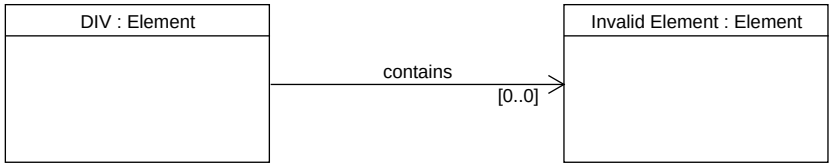


Figure A.2: Usability Experiment: Assignment 1.2

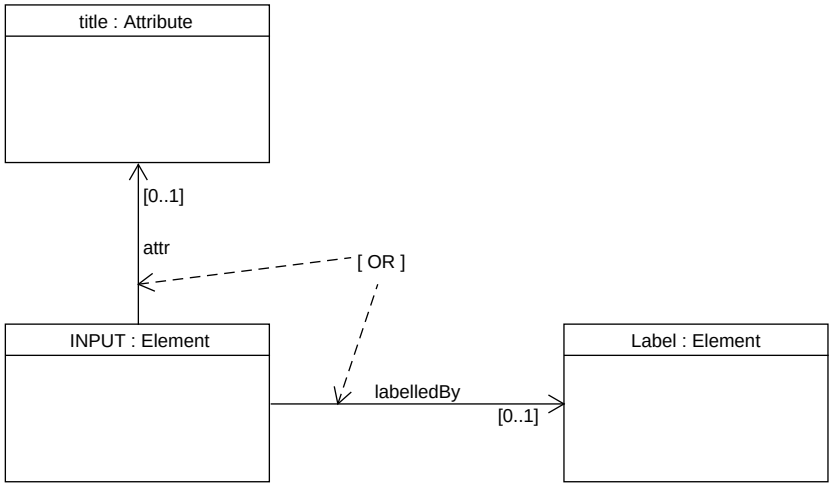


Figure A.3: Usability Experiment: Assignment 1.3

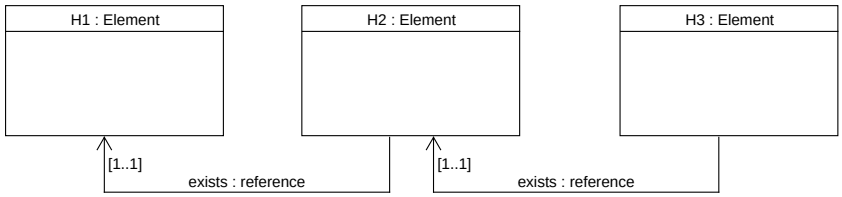


Figure A.4: Usability Experiment: Assignment 1.4

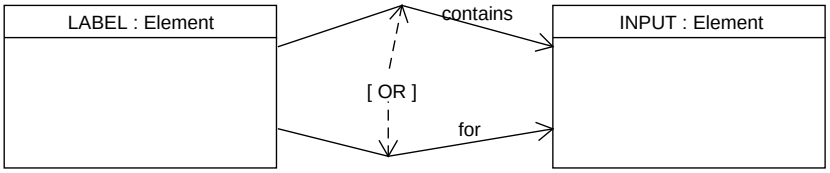


Figure A.5: Usability Experiment: Assignment 1.5

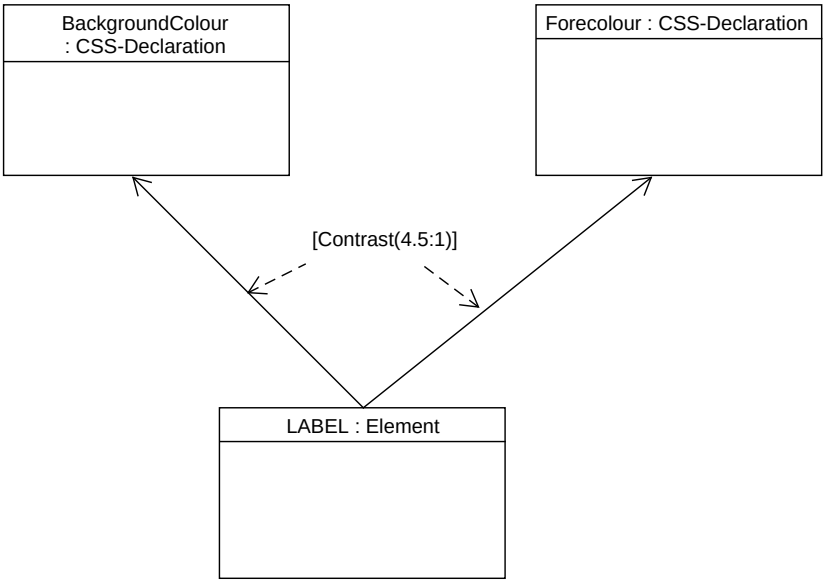


Figure A.6: Usability Experiment: Assignment 1.6

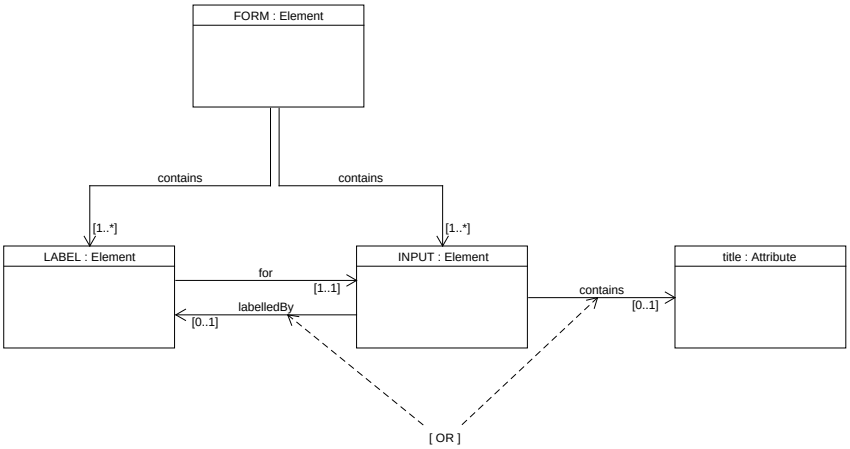


Figure A.7: Usability Experiment: Assignment 1.7

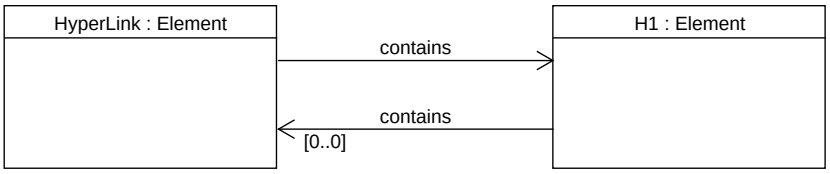


Figure A.8: Usability Experiment: Assignment 1.8

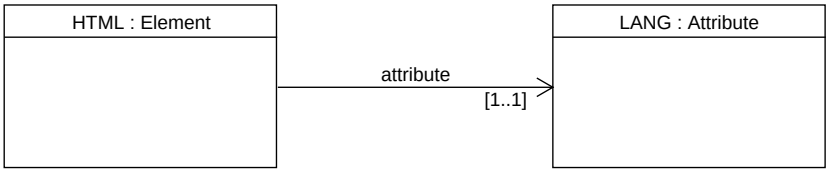


Figure A.9: Usability Experiment: Assignment 1.9

Appendix A

Models from Part 2

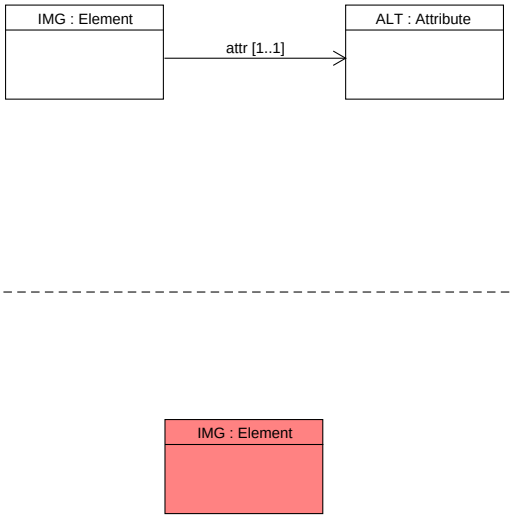


Figure A.1: Usability Experiment: Assignment 2.1

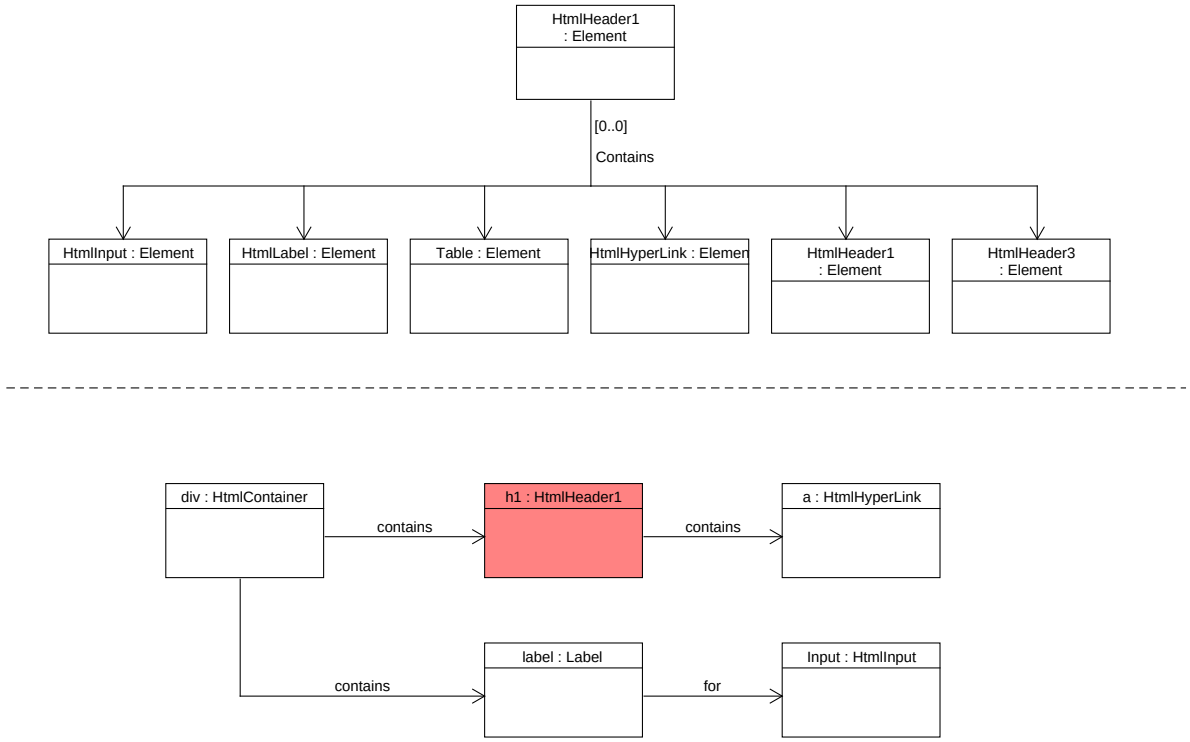


Figure A.2: Usability Experiment: Assignment 2.2

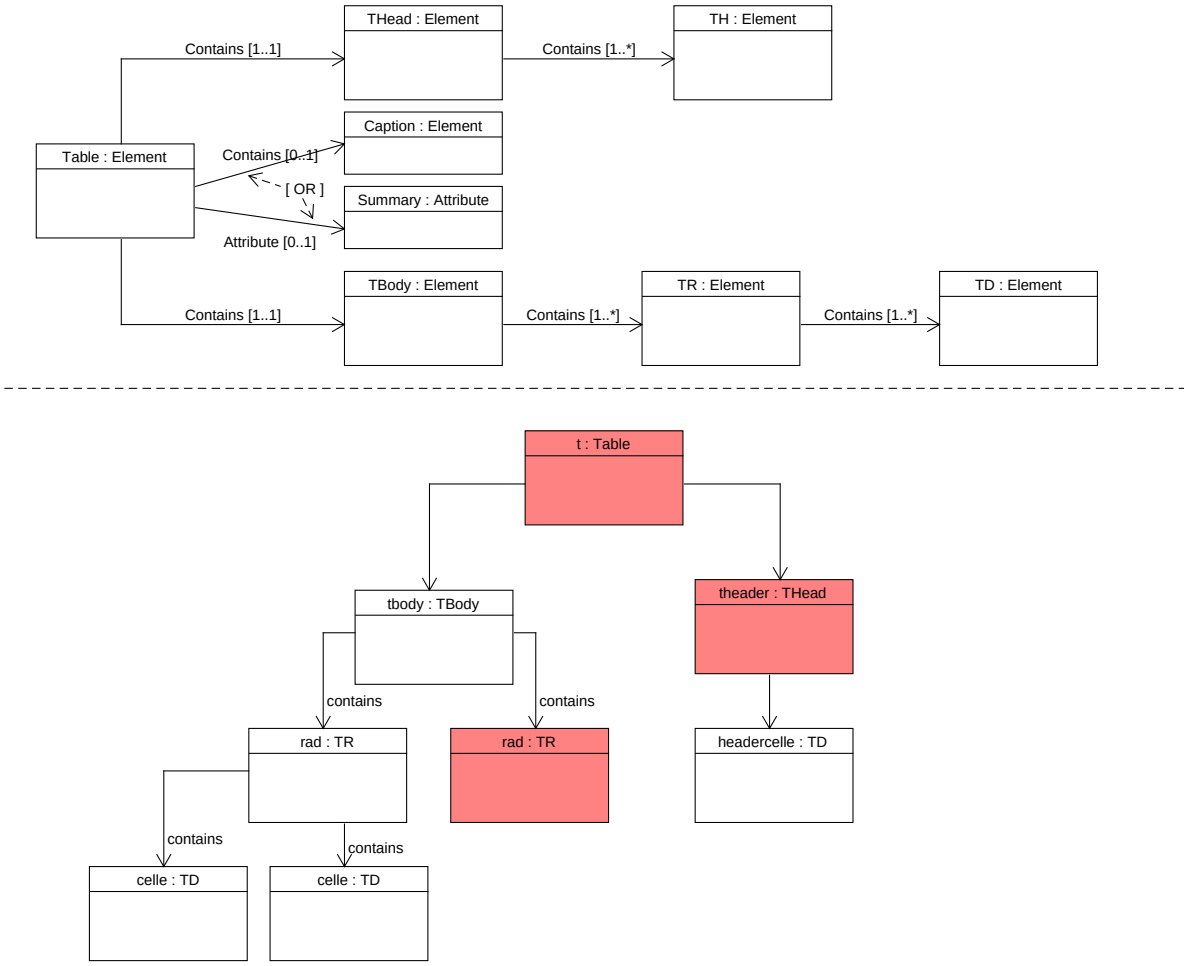


Figure A.3: Usability Experiment: Assignment 2.3

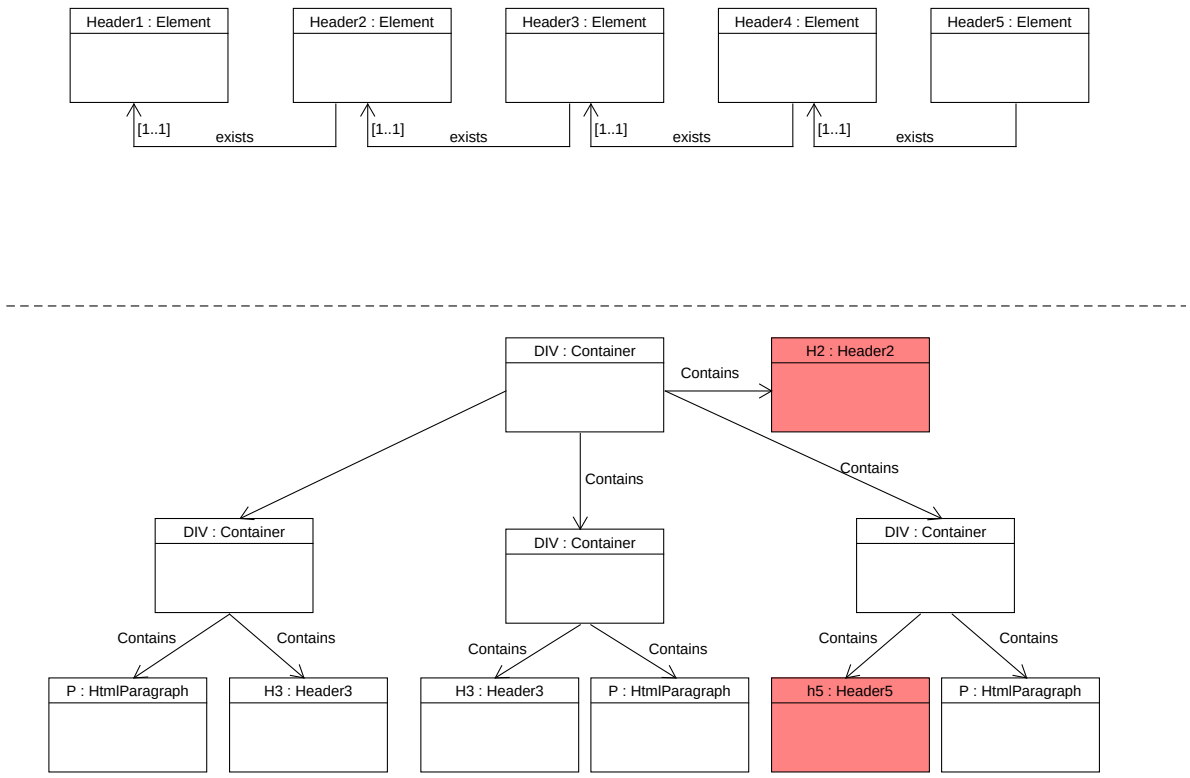


Figure A.4: Usability Experiment: Assignment 2.4

Appendix A

Models from Part 3

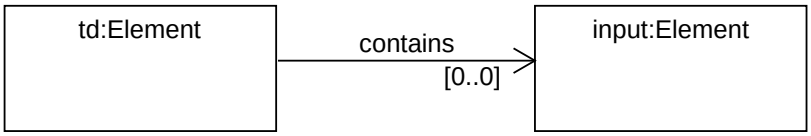


Figure A.1: Usability Experiment: Assignment 3.1

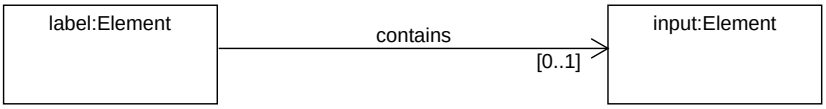


Figure A.2: Usability Experiment: Assignment 3.2

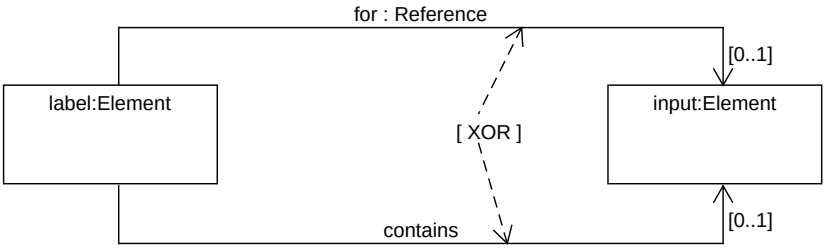


Figure A.3: Usability Experiment: Assignment 3.3

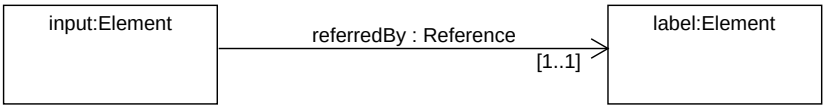


Figure A.4: Usability Experiment: Assignment 3.4

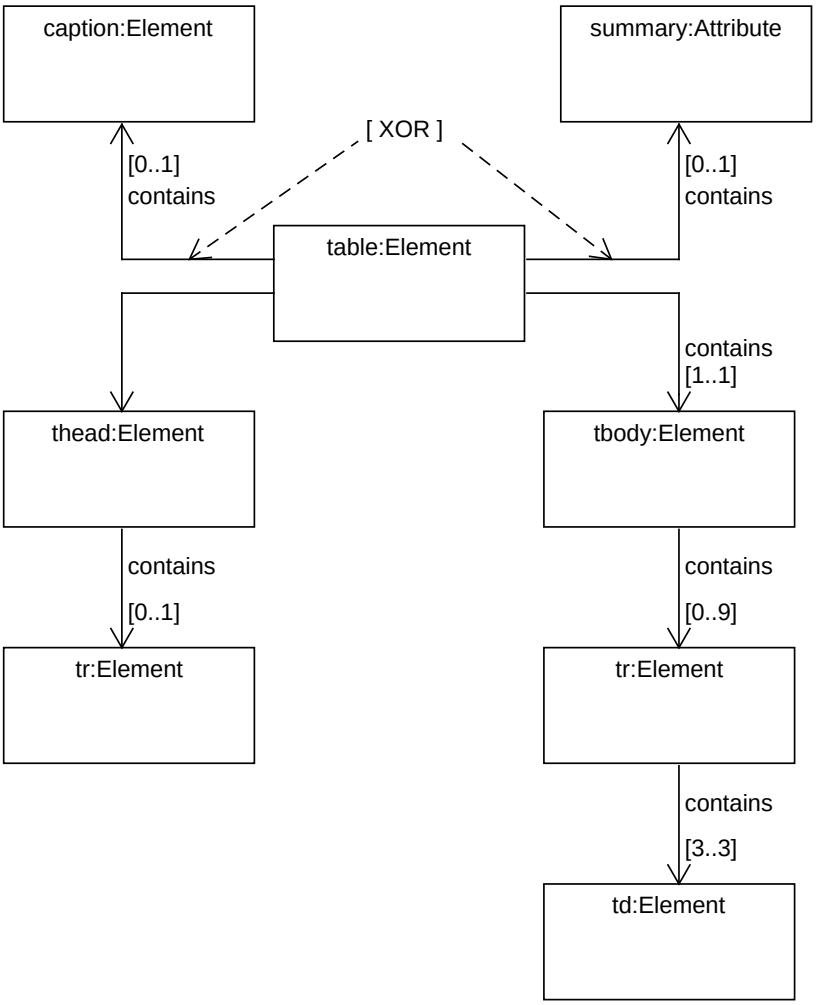


Figure A.5: Usability Experiment: Assignment 3.5

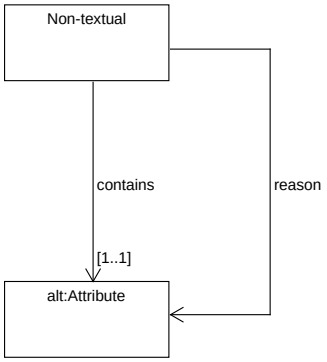


Figure A.6: Usability Experiment: Assignment 3.6

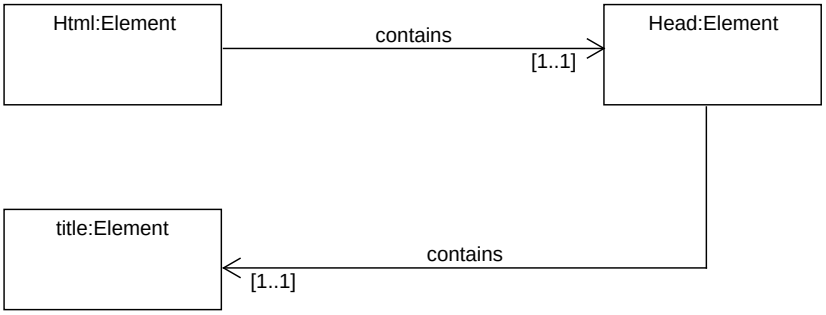


Figure A.7: Usability Experiment: Assignment 3.7

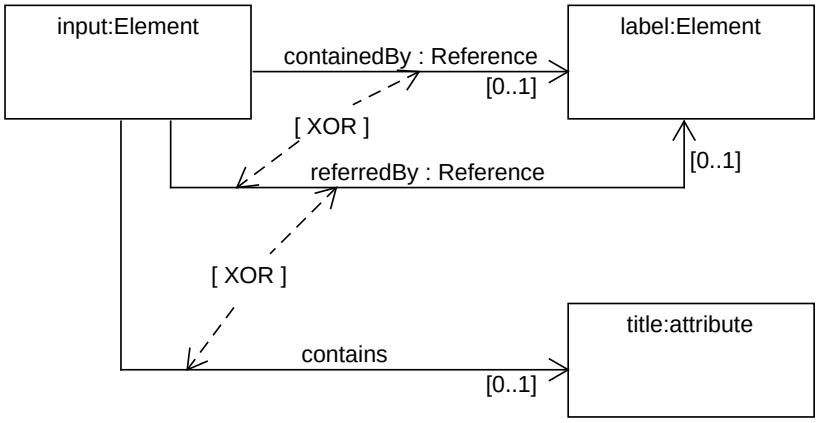


Figure A.8: Usability Experiment: Assignment 3.8

Bibliography

[1] AChecker. Achecker: Web accessibility checker. http://achecker.ca/checker/index.php. Accessed: 2016-04-28.

[2] ACOS. Acos as. https://www.acos.no/. Accessed: 2016-04-28.

[3] Alesund. Alesund kommune. http://www.alesund.kommune.no/. Accessed: 2016-04-28.

[4] Barvik, P. and Lamo, Y. (2013). Model to model transformation tool for the dpf workbench.

[5] Bergen. Bergen kommune. https://www.bergen.kommune.no/. Accessed: 2016-04-28.

[6] Blindeforbund, N. Norges blindeforbund. https://www.blindeforbundet.no/. Accessed: 2016-05-08.

[7] Bork, M., Geiger, L., Schneider, C., and Zündorf, A. (2008). Towards roundtrip engineering-a template-based reverse engineering approach. In *Model Driven Architecture–Foundations and Applications*, pages 33–47. Springer.

[8] Bos, B., Çelik, T., Hickson, I., and Lie, H. W. (2005). Cascading style sheets, level 2 revision 1 css 2.1 specification. *W3C working draft, W3C, June*.

[9] Brajnik, G. (2004). Comparing accessibility evaluation tools: a method for tool effectiveness. *Universal Access in the Information Society*, 3(3-4):252–263.

[10] Brambilla, M., Cabot, J., and Wimmer, M. (2012a). *Model-driven software engineering in practice*, volume 1. Morgan & Claypool Publishers.

[11] Brambilla, M., Cabot, J., and Wimmer, M. (2012b). Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):8.

[12] Bruneliere, H., Cabot, J., Dupé, G., and Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032.

[13] Centre, T. B. R. The brc and our vision. https://www.brreg.no/about-us/. Accessed: 2016-05-03.

[14] Centre, T. B. R. Elmer: User interface guidelines for governmental forms on the internet, version 2.1. http://brukskvalitet.brreg.no/elmer21/elmer_2_1_english.pdf. Accessed: 2016-05-03.

[15] Chikofsky, E. J., Cross, J. H., et al. (1990). Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17.

[16] Darwin, P. B. and Kozlowski, P. (2013). *AngularJS web application development*. Packt Publ.

[17] DIFI. Universell Utforming kontrast. http://uu.difi.no/veiledning/nettsider/uu-skolen/kontrast. Accessed: 2015-09-22.

[18] DIFI. Universell Utforming skjema. http://uu.difi.no/veiledning/nettsider/uu-skolen/kontrast. Accessed: 2015-09-22.

[19] DIFI. Universell Utforming tekst og struktur. https://uu.difi.no/artikkel/2015/07/tekst-og-struktur. Accessed: 2016-02-14.

[20] Ernst, J. What is metamodeling, and what is it good for? http://infogrid.org/trac/wiki/Reference/WhatIsMetaModeling. Accessed: 2016-01-14.

[21] Fawcett, J., Ayers, D., and Quin, L. R. (2012). *Beginning XML*. John Wiley & Sons.

[22] Files, M. C. (1999). Object constraint language.

[23] for Public Management, T. A. and eGovernment. About difi. https://www.difi.no/om-difi/about-difi. Accessed: 2016-05-03.

[24] for Public Management, T. A. and eGovernment. It i praksis 2015. https://www.difi.no/artikkel/2016/02/it-i-praksis-2015. Accessed: 2016-05-03.

[25] for Public Management, T. A. and eGovernment. Kva er universell utforming? https://uu.difi.no/kva-er-universell-utforming. Accessed: 2016-05-03.

[26] for Public Management, T. A. and eGovernment. Kvalitet paa nett. http://kvalitet.difi.no/kriteriesett/print. Accessed: 2016-05-03.

[27] for Public Management, T. A. and eGovernment. Universell utforming, statusmaaling 2014. https://www.difi.no/artikkel/2016/04/universell-utforming-statusmaling-2014. Accessed: 2016-05-03.

[28] Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

[29] Gay, G. and Li, C. Q. (2010). Achecker: open, interactive, customizable, web accessibility checking. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*, page 23. ACM.

[30] Gerber, A. and Raymond, K. (2003). Mof to emf: there and back again. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 60–64. ACM.

[31] GROUP, O. M. et al. (2014). Object management group model driven architecture (mda) mda guide rev. 2.0.

[32] GROUP, O. M. et al. (2015). Meta object facility (mof) 2.5 core specification.

[33] Hickson, I. and Hyatt, D. (2011). Html5. *W3C Working Draft WD-html5-20110525, May*.

[34] Kruchten, P. (2004). *The rational unified process: an introduction*. Addison-Wesley Professional.

[35] Metamodel, K. D. Url: http://www.omg.org/spec. *KDM/1.1*.

[36] of Bergen, U. C. Webdpf editor. http://data2.hib.no:8080/dpf2.0/index.htm. Accessed: 2016-04-28.

[37] Omg (2000). Meta object facility (mof) specification. *Meta Object Facility (MOF) Specification*.

[38] Os. Os kommune. https://oskommune.no/. Accessed: 2016-04-28.

[39] Oslo. Oslo kommune. https://www.oslo.kommune.no/?lang=no_NO. Accessed: 2016-04-28.

[40] Pérez-Castillo, R., de Guzmán, I. G. R., and Piattini, M. (2010). Architecture-driven modernization. *Modern Software Engineering Concepts and Practices: Advanced Approaches: Advanced Approaches*, page 75.

[41] Pérez-Castillo, R., De Guzman, I. G.-R., and Piattini, M. (2011). Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532.

[42] Pérez-Castillo, R., García-Rodríguez de Guzmán, I., Piattini, M., and Places, A. S. (2012). A case study on business process recovery using an e-government system. *Software: Practice and Experience*, 42(2):159–189.

[43] PowerMapper. Powermapper: one click website tools. http://www.powermapper.com/products/sortsite/. Accessed: 2016-04-01.

[44] PowerMapper. Sortsite, trial. http://www.powermapper.com/products/sortsite/try/. Accessed: 2016-04-07.

[45] RFP, C. F. Object management group. *OMG PTC Document realtimel*, pages 97–O5.

[46] Ringsaker. Ringsaker kommune. http://www.ringsaker.kommune.no/. Accessed: 2016-04-28.

[47] Rugaber, S. and Stirewalt, K. (2004). Model-driven reverse engineering. *Software, IEEE*, 21(4):45–53.

[48] Rutle, A. (2010). Diagram predicate framework: A formal approach to mde.

[49] Sánchez-Cuadrado, J., De Lara, J., and Guerra, E. (2012). *Bottom-up meta-modelling: An interactive approach*. Springer.

[50] Seidewitz, E. (2012). Uml 2.5: Specification simplification. In *Third Biannual Workshop on Eclipse Open Source Software and OMG Open Specifications*, volume 82.

[51] Software, P. Sortsite 5 user manual.

[52] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422.

[53] Tilley, S. and Huang, S. (2001). Evaluating the reverse engineering capabilities of web tools for understanding site content and structure: A case study. In *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, pages 514–523. IEEE.

[54] Tonella, P. (2005). Reverse engineering of object oriented code. In *Proceedings of the 27th international conference on Software engineering*, pages 1–3. ACM.

[55] W3C. Web accessibility initiative. https://www.w3.org/WAI/intro/w3c-process.php. Accessed: 2016-03-10.

[56] W3C et al. (2008). Web content accessibility guidelines (wcag) 2.0. https://www.w3.org/TR/WCAG20/. Accessed: 2016-02-13.

[57] W3C et al. (2014). Html5. https://www.w3.org/TR/html/. Accessed: 2016-05-30.

[58] WCAG2.0 and WCAG2.0. WCAG2.0 luminance formula. https://www.w3.org/TR/2008/REC-WCAG20-20081211/#relativeluminancedef. Accessed: 2015-02-16.

[59] WebAIM. Wave, web accessibility evaluation tool. http://wave.webaim.org. Accessed: 2016-04-07.