Elgar de Groot
Stijn van Schooten

Technical University of Delft
Faculty of Computer Science

Bachelor Thesis:
A model driven IDE for M-industries’ Alan.

A bachelors thesis about Model Driven Design
& Language Design

Client: Corno Schraverus  M-industries
Coordinator: Martha Larson  Delft University of Technology
Coordinator: Felienne Hermans  Delft University of Technology

September 3, 2015
Acknowledgements

We would like to acknowledge the provider of this opportunity and developers of the platform we used at M-industries, especially Corno Schraverus, Joost-Wim Boekesteijn, Gerben Kunst and Gerbert van Nieuwaal for their support and guidance. We also thank the other employees at the company for their cooperation and help with solving some of the difficult problems we encountered during this project. Special thanks to our project counselors Felienne Hermans and Martha Larson, without their guidance and encouragement we would not be able to finish this project. Last but not least, we would like to express out gratitude to our friends and family for their support and willingness to listen to our ramblings.
Abstract

While software is becoming increasingly important in our world, software development is also advancing with an increasing pace. One of the reasons is the increase in available information, which triggered the birth of a new programming paradigm: Model Driven Development (henceforth MDD). Though this can shorten the development time and make it easier, there is no real support for this method, and also no fully developed environment. This is where M-industries’ Alan is key. A new MDD platform, but with no editor support. That was the initial scope of our project: create an editor capable of supporting Alan.

This was no easy task, and so some preliminary research was done, which evaluates existing web-based editors based on the requirements set by us. The main development phase consisted of agile programming cycles where the targets and tasks were subject to changes. This enabled us to focus on creating an intrinsically correct system instead of a fully featured one that needs a lot of patching and cleaning.

The result was a well rounded, integrated IDE that has powerful Alan specific features, but may lack some more basic editor features. The IDE was not only developed to aid in model driven development, but was actually developed itself in a model driven way, using M-industries’ platform. This allowed for a deep integration with Alan, where the language definition became part of the IDE. To do this, the concept of an ”editor state” was introduced, which proved to be an essential an powerful concept for creating an editor for model driven development.
Contents

Introduction 1

1 Project analysis: An editor for M-industries 2
  1.1 Project description and analysis 2
  1.2 Requirements 2
    1.2.1 Functional requirements 3
    1.2.2 Quality requirements 3
    1.2.3 Definition of success 4
  1.3 Motivation 4
  1.4 Development methodology 4
  1.5 Situation analysis and projection 5

2 System Design 6
  2.1 Generic Description 6
  2.2 Document Structure 7
    2.2.1 Code Folding 7
  2.3 Embedded Alan 8
  2.4 Lexing 8
  2.5 Internal Parsing 8
  2.6 Reference Resolving and Jump-to-Definition 9
  2.7 Syntax Highlighting 10
  2.8 Auto-completion and Suggestions 11
  2.9 The IDE wrapper 12

3 Process analysis 13
  3.1 Overview of major milestones 13
  3.2 The choice for Alan 14
  3.3 Editor state model 14
  3.4 Projects and IDEs 15
  3.5 An interactive environment 15
  3.6 Editability and partial parsing 15

4 Testing 17
  4.1 Testing in Model Driven Development 17
  4.2 Testing the lexer 18
  4.3 Testing the parser 18
  4.4 Testing the reference resolver 19
  4.5 Testing the user interface 19
  4.6 Performance testing 19

Conclusion 21

Discussion and Recommendations 22

A M-industries 24
Introduction

This thesis is part of our final bachelor project, which was commissioned by the company M-industries. The assignment was to create a programming environment for their own language, preferably using their own technologies as a basis. This report will describe the most important and interesting parts of the process, along with some background on both the company as well as the technologies and methods used.

To fully understand this report, the assignment and its analysis must be read, which can be found in chapter 1. This will detail what was interpreted of the assignment, what the set requirements were (both functional and quality wise), but also what development methodology was intended to use and how it was intended to be applied. And finally what our impression of the current situation was and a prediction of what the impact of this project would be.

Chapter 2 contains all the information needed to understand the technical aspects of this project, the design and also provide background on some decisions that were made. A generic system description can be found here, along with more detailed descriptions of several key parts. Some of these parts require a more substantial explanation on why that choice was made, this explanation can be found in chapter 3, which contains detailed explanations of why certain design or technical choices were made.

All these implementations and choices have been tested in a way that fits the programming paradigm and system facet, which are elaborated in chapter 4, where all the testing methods and the projected consequences are explained.

This report summarizes a long road on which a large number of choices have been made, which came together in the existence of an (almost completely) functional IDE. The most powerful aspects of M-industries’ Alan language have been used, which resulted in improvements in the Alan language itself too.
1. Project analysis: An editor for M-industries

The question “why reinvent the wheel?” might come to mind when reading this report, since editors for specific languages have been available since the dawn of programming. But with each new paradigm, there arose a new Integrated Development Environment (henceforth IDE) with a new workflow that enhances the programming experience. This chapter will detail the assignment, as it was given, but also explain why the scope and focus of the project has shifted and whereto. Following are the requirements and the definition of success for this project, which has changed also, with a more generic motivation for the new assignment after that. Finally the development methodology will be discussed, what was intended and what was executed, with an situational analysis and projection based on the company as a final section.

1.1. Project description and analysis

The first impression of the assignment was to exactly not do that, do not reinvent the wheel and use an existing editor, so an environment tailored to the needs of Alan could be created (the language M-industries has developed). But the first projections were way off since the initial assignment changed a large amount, which also resulted in the longer project duration and postponing this thesis and presentation. The first two weeks consisted of a research phase, which was needed to deepen our understanding of the problem and its facets. Also, the research phase was used to do some trials with proposed methods and platforms. A more detailed evaluation of the research phase can be found in the report in appendix E, this will also include how the insights from this week have influenced the continuing of our work. For the exact description of the project as was given can be found in appendix B.

The original project described the building of a fully functional editor, using existing techniques. This proved to be possible, but very complex, since the whole range of Alan derivatives should be supported. The actual insight that, since Alan was self defining, using their own platform would make it easier to manage the complexity. With this knowledge in mind we reformulated the assignment to be more of a research project, where the final product would more of a proof of concept instead of functional editor. The new assignment was to find out what was needed to create an that could use Alan to lex, parse and display a Alan document. This meant finding out how the usual IDE features would have to be realized by using Alan. To understand how Alan is fundamentally different from regular programming languages, please read appendix C

1.2. Requirements

The requirements were bound to change since the beginning, since the solutions needed to design and build an IDE for them were not apparent and their platform was constantly shifting too. This meant that constant re-evaluation was needed, and going along with these shifts was important to not make the IDE deprecated the moment the project was finished. This
section will exclusively detail the final requirements, since the initial ones can be found in the research report and project description (appendices E and B respectively).

1.2.1. Functional requirements

The IDE would be used to improve development speed and quality for the company, and the requirements were also based on these prospects. The most important functionalities are the ones that other editors do not provide, therefore the language specific features are the most valuable. Very basic IDE functionalities such as code editability have therefore less priority than a jump-to-definition functionality or context aware suggestions. Below is a list of functional requirements sorted on priority.

- The IDE supports the Application model language.
- The IDE has support for projects.
- The IDE displays error messages provided by parser engine.
- The IDE displays error messages of unresolved references.
- The IDE supports the jump-to-definition feature.
- The IDE can show suggestions based on the language grammar.
- The IDE supports code folding.
- The IDE supports syntax highlighting.
- The IDE supports any language designed with Alan.
- The user can insert suggestions in the document (code completion).
- The IDE supports the jump-to-meta-definition feature.
- The IDE supports editability of a document.

1.2.2. Quality requirements

Besides functional requirements, quality requirements are needed to define the goal more clearly. A functionally working system that responds too slow is not usable in practice and therefore not successful. Other quality requirements could describe aesthetics or memory usage. These usually are more vague, since they are more susceptible to interpretation and rely on the user feeling that the software is adequate for its purpose.

As for this project, the most obvious quality requirement was the responsiveness of the editor and how instinctively the use is. Since the system would be web-based for this project, the layout and styling of the editor could be trivially modified with some simple modifications. Therefore, the biggest point of focus was functionality and responsiveness. Section 2.9 details the problems we encountered in a later stage with real life example projects.

Since the whole model would be loaded into the memory, we knew from the start that RAM usage could become a problem with large projects. The complexity of the file does not matter in this case, since Alan is a context free language, eliminating that factor from the list of item to be evaluated and tested. Below is a list of further quality requirements:
\begin{itemize}
\item The IDE displays the code correctly.
\item The IDE has a clear visual interface.
\end{itemize}

\section*{1.2.3. Definition of success}

The success of our project can not strictly be measured by the number of requirements (both functional and quality wise), but can be, more or less, derived from these. Therefore success can be achieved in multiple ways, and the actual definition relies on a generic functionality since this is more of an exploratory project for providing IDE functionalities in a model driven manner.

In this case the editor could be called a success if it is able to load a real-life size project, lex and parse it correctly, and resolve the resolvable references. If one of these steps fails, it should show where and why it failed, optionally with a suggestion. Finally display the document in a fashion the user can read the file and see/understand the function and use of the tokens. Reaching more goals would make the editor more usable, thus making the project more successful.

\section*{1.3. Motivation}

While the most trivial aspect of motivation for our project is, by default, to finish the last part of our bachelors degree, the most important aspect is the one in which your stance towards the company or target audience is defined. This would initially be to help them develop faster, and gain/provide better insight in what kind of impact the structure of Alan has of current editors. When the choice was made to switch to an MDD approach using Alan, this aspect expired and a more proof-of-concept point of view was adopted. The focus of the project shifted from creating a finished product to researching and developing an IDE for Alan with as much of the features the regular editors would have, but built by using an MDD approach. This shift was caused by choosing for the Alan platform, and the fact that no real editing environments have been built using MDD techniques.

\section*{1.4. Development methodology}

Using a platform where most of the algorithmic code is generated from a schema (a schema (plural schemata or schemas) describes an organized pattern of data or behavior that organizes categories of information and the relationships among them.) means that one of the biggest time savers is the fact that modifying the internal structure uses only a few lines of code and modifications down the line where needed. This method of low-impact changes on the underlying information invites for a very agile workflow, meaning that numerous but small milestones will be defined, called sprints. This increases the productivity since there is more focus on one single aspect in per sprint, but also the overhead and fault in time projections will be reduced since these do not snowball through the rest of the project.

Only the final targets are defined, creating a large body of possible paths to reach your targets. Certain target based projects will benefit greatly, like ours. Since the programming platform makes a agile workflow easier to work with, this become the method of choice within the company. This workflow should also be reflected upon in the design of the IDE, supporting the making of various small changes.
During the course of this project, multiple redefinitions of the current milestones have taken place, since obstacles were encountered that needed more of a supporting structure to define the correct approach to solve it. This meant that, while already maintaining a agile flow, the sprints were variable of length depending on the facet that needed to be done at that time. Using agile and sprints has helped enormously with coping with the many modifications to the platform, requirements and our own vision of the IDE.

1.5. Situation analysis and projection

Before starting the project, M-industries had to work with regular editing programs, like Netbeans\[7\] or PhpStorm\[5\] with simple syntax rules to support syntax highlighting and adding scripts to parse the files. This means a custom setup very time an editor was made to work with Alan. This environment was not only archaic, it was also only used by a few of the developers, leading to a undocumented and unmanaged collections of syntax definitions and scripts.

The above mentioned editors are designed to assist with developing other than MDD, meaning support is faulty and incomplete (if existent at all). Designing and writing the Alan structures required a full understanding of the language and possibilities. These structures started out in simple JSON (JavaScript Object Notation, see [3]), enriching the document while parsing, whereas this has grown out to a fully fledged DSL (Domain Specific Language, a programming language, usually a derivative, designed to solve a certain type of problems easier), which can be read and verified with the toolchain M-industries has developed.

This is where the IDE could prove to be most useful. Since Alan is only a base language, designed to be derived into more specific DSLs. Using Alan as a basis of the IDE could prove to solve the problems with providing support for MDD languages. Because there was no solid basis upon more research could be done into features that could support MDD specifically, this MDD based IDE could turn out to jump start more advanced developing support methods based on Alan and M-industries vision on MDD.
2. System Design

In this section the design of the system and the different functionalities of it will be described. All the separate parts and aspects of the editor state model will be analyzed, explaining which subsystem is responsible for mutating that part of the editor state and which functionalities it enables. This will also be the first introduction into the thought process of the new MDD programming paradigm (for more explanation see appendix C). After a generic description of the system, the main document structure will be examined and how code folding is implemented. After that the coupling of the editor state model with the Alan language definition is explained, along with the incorporated lexer and parser. Thereafter the jump-to-definition feature, syntax highlighting and auto-completion functionalities are discussed. Finally, we will see how the editor is wrapped into an IDE to enable Alan projects and how internal settings are implemented in the IDE.

2.1. Generic Description

![Component diagram of the IDE.](image)

The main function of the IDE is to aid in the development of applications for clients. This means that support for the Application language is one of the most important features. Halfway through the project the Application language was split into multiple definitions and an application was now defined in a project definition. The user can load a project into the IDE. This can be either a new project or an existing one. Once the project is loaded, the file structure will show on the left side of the IDE, with the name of the project shown above. When clicked on a file, it will be lexed and parsed, the references resolved and displayed on the right. If there are any errors during lexing, parsing or reference resolving, they will be displayed in the field below where the file is displayed.

The system design is divided in two parts: the main IDE model, which incorporates the editor state model, and the interface model. The first part models the entire IDE. When
the IDE is started, an instance of this IDE model is initialized. The IDE model effectively describes every possible state the IDE can be in. The second part is purely for the interface. It depicts how each element of the IDE model is to be rendered, if it should have a callback function, etcetera. These two models together describe the whole IDE, which states it can be in and how it should be displayed.

The largest part of the codebase is for actually mutating an instance of the IDE model. The IDE model depicts what is possible, but there has to be code to actually do the work. The processes that mutate the editor state include the lexer, parser and reference resolver. Figure 2.1 gives a rough schematic representation of the components in the system.

### 2.2. Document Structure

![Diagram of the document structure in the editor state model.](image)

The structure of the document in the editor is based on lines and blobs. The editor contains a set of lines, ordered as a kind of doubly linked list. There is a reference to the first line and a reference to the last line. Every line holds a reference to its predecessor and its successor. Furthermore, each line contains a reference to a parent line, which is based on the indentation of the line. A line contains a set of blobs, ordered in the same doubly linked list style as the lines. The blobs represent groups of characters. For example, a blob can represent a space, a tab, a comment or a token, but more on this further in the chapter.

It is the lexer that brings the structure in the document. Along with classifying different kinds of blobs, it creates new blobs, new lines and the correct references between them. When lexing a new line, it also keeps track of the amount of tabs at start of the line to determine the indentation of the line. According to the indentation and that of the preceding lines it determines which line should be the parent line.

#### 2.2.1. Code Folding

The references to the parent lines is done to enable code folding. If a line has children it will be labelled as foldable and a fold button will be rendered in the sideline of the editor. If the fold button is clicked, all the children will be recursively hidden, and if clicked again they will be shown again.
2.3. Embedded Alan

A key aspect of the editor state is the integration of the Alan language definition models into the editor model. This has made it possible to keep the different components of the system more independent, even though it makes the editor model more complex. In the original situation the lexer, parser and API generator (see Appendix D.3.3 for more on these components) were all dependant on the output of the former and on the language definition. When developing the editor model we noticed that we were essentially incorporating the output of these operations into the model. So instead of keeping the output of the lexer and parser separate from updating the editor model, these components were modified so that they directly mutate the editor state. Because it is unnecessary to generate an API for the IDE, the API generator was discarded in favour of a custom reference resolver. So, by incorporating the grammar and schema models in the editor state, the only dependency of the custom lexer, parser and reference resolver is the editor state.

2.4. Lexing

The custom lexer splits the document in lines and blobs. When a document is first opened, a new editor instance is created with a single line and a single ‘raw blob’. The raw text of the document is copied into the raw blob and the lexer starts its work, inserting new blobs preceding the raw blob and new lines after the initial line. Originally, the lexer would discard all the spaces, tabs, comments and newlines but for visualization this is valuable information so it is included in the editor state model. The remaining (majority) of the text is classified as tokens. Tokens come in three flavours: keywords, numbers and literals, and are the only blobs that the parser is concerned about.

If the lexer encounters an error in the document, it keeps the remaining text in the raw blob, invalidates it and attaches an error message to it.

2.5. Internal Parsing

The custom parser has three main functions: it validates the tokens using the grammar, it further categorizes the literals in one of the five types as shown in figure 2.4 and it inserts node blobs into the editor state. The validating provides valuable error information. If an error occurs during parsing, the faulty token is invalidated and an error message is inserted. The error message can then be shown in the error message field below the document view. The
error messages will include a list of suggestions with all the valid options. The categorizing of the literals allows for more detailed syntax highlighting to aid the user in understanding the code. See section 2.7 for more on syntax highlighting.

The node blobs that are inserted into the editor state during parsing are essential for the following step: reference resolving. Originally, the parser would generate a raw language instance in JSON format. This file contains a lot of information that cannot be deduced from the tokens alone, but is necessary for reference resolving. Therefore it was necessary to include the missing pieces, i.e., the node blobs. For a representation of the editor state model including the parsing step see figure 2.4.

Another thing for which the parser is responsible and which is necessary for reference resolving, is linking the tokens and node blobs to the language grammar (not shown in figure 2.4). Also, the parser creates a list of pointers to all the references in the document that it encounters. While this is not strictly necessary, it saves an extra iteration over the editor state.

2.6. Reference Resolving and Jump-to-Definition

Most of the preparatory work for reference resolving is done during parsing. After a document is loaded and parsed, the list of references is iterated over and for every reference the reference resolver is called. The reference resolver checks if the reference is not already resolved and looks up its definition in the language schema via the language grammar. Remember that the parser has linked the reference to its grammar rule and that the grammar is linked to the language schema (see figure 2.1). In the language schema the reference path is defined. The
The reference resolver can follow this path through the node blobs and references in the editor state until it arrives at the node that the reference refers to. If it succeeds, the reference is validated and a pointer is saved to the referred node. If it does not succeed, because for example the key does not exist, the reference is invalidated and an error message is created. The error message will include, if it is possible, a list of suggestions. If as in the previous example a key cannot be found, the list of existing keys will be displayed.

The references that have been resolved will be underlined and can be clicked on, after which the blob to which the reference refers will be highlighted so that the user can see it in a glance. This is a powerful feature, since otherwise the user would have to follow the reference step by step himself, which can be a very cumbersome task especially in large documents and with long references. This feature is called 'jump-to-definition', because a reference points to a node somewhere in the document and if clicked on the reference the place where the node is defined lights up.

### 2.7. Syntax Highlighting

Syntax highlighting evolves naturally from the editor state model. The more types that are distinguished in the model, the more specific syntax highlighting is possible. By coupling a styling to each type and state of blob it is possible to create a whole range of syntax colourings. Because the styles are coupled to specific states of the editor, if the editor state changes, the syntax highlighting automatically changes as well. Figure 2.6 shows the styles that are coupled to the different states and types in the editor state model. It can be seen...
that if the document is only lexed the syntax highlighting distinguishes four colours (including comments), but when it also has been parsed there are already eight colours. Note that the syntax highlighting is language dependant, since literal tokens can be classified into either of the five types depending on the language definition.

2.8. Auto-completion and Suggestions

The editor state model with the incorporated grammar enabled the development of auto-completion functionality in the editor. While full editability of code would make the feature even more powerful, it shows the power of the editor state. The main functionality of the feature is that it shows what valid tokens may be entered after the caret position in the editor, or what alternatives may be entered instead of the token the caret is in. With a key-combination the list with suggestions can be summoned, and optionally one can be chosen and automatically entered by the editor.

The functionality works because in the editor state model the node blobs are introduced and the grammar is linked to the node blobs and tokens. Provided that the document is parsed up until the caret position in the document, the last token or node blob can be traced back, its definition can be looked up in the grammar and from there the following valid token can be determined. This can be shown to the user, and after the user has made a choice, a token can be created and inserted into the editor state. This can potentially be repeated without the user having to type anything, except when a literal is chosen, effectively allowing projectional editing. If the editor is used in this way, parsing would be unnecessary, because every addition to the document is already sure to be valid.

This functionality can be further extended by including reference resolving for reference, sparse matrix key and dense matrix key tokens so that suggestions for these can be shown as well. The only typing that then would be necessary is for dictionary keys and text literals (see the Discussion and Recommendations for further discussion of this functionality).
2.9. The IDE wrapper

Having a lot of functionalities that could support writing Alan is very nice, since these concepts rely on intricate linguistic concepts. But these have no direct effect in support features, and need a graphical interface to bind to these features. While this sounds very logical, it also introduces a lot of difficulties in completing all the milestones (see chapter 3), since interface work is error prone and requires work to be done on multiple aspects at the same time: the engine has to support a function, which has to be bound to an interface element that has to be rendered correctly and events have to be handled when that element is triggered. Put a lot of these GUI (Graphical User Interface) elements together and you have created a usable shell or wrapper around our editor state model, you have created an IDE.

As explained, this IDE is not much more that a wrapper around multiple editor states, providing a graphical interface for handling projects and providing methods of loading and saving these projects. But this also meant that loading a project enables you to load all the files in that project in separate editors, making the IDE lex, parse and resolve each one of them. This uses a lot of computational resources, so it would be advisable to work on the least amount of files simultaneously. After measuring the footprint while lexing and parsing a large file (so large, parsing it makes the browser crash), some major improvements were made to the language and especially to the way contexts are handled. This resulted in a major speedup, which was followed by improving the way the markup was translated and rendered. Separating or “chunking” a large project into multiple smaller linked files is common practice in software development, since this improves readability and reduces complexity. This results in a cluster of Alan files that have external references (that can not be resolved independently), but also reduces the processing time per document. A graphical representation of these speedups can be found in appendices H.11 and H.12.

To work with these projects an interface was needed that looks and feels like an conventional editor, a graphical representation of the editor state model, which is actually bound to that instance. To help explain these concepts, an screenshot can be found in appendix H.13. The large field with the text is the editor, this would be the place where text would me written and modified. Basic editing of this text is possible, but has limited editing methods (see section 3.6). The gray area beneath this editor is the output box, which would contain errors, suggestions and basic info on the token that is selected at that moment. Clicking the erroneous word (which is a reference to the target word in the document) would make the caret jump to that word, without the need for complex mechanics. To the left a project area can be seen, with the current project name at the top, project related buttons underneath the name (from left to right: save application, load project, import project, open settings pane) and finally the file tree, showing how the file structure where the Alan files can be opened in an editor. The settings pane is not shown in this image, but slides out from the left and contains some controls for that project. These are: project name, autosave interval (in seconds, how much time is between saves to the server), autostore interval (in seconds, how much time is between storing the project in localStorage and/or sessionStorage), enable/disable localStorage (if autostore should store the project in the in-browser localStorage, which is persistent), enable/disable sessionStorage (if autostore should store the project in the in-browser sessionStorage, which is non-persistent) and tab width (in spaces).
3. Process analysis

Though concrete functionality might seem to be the most important information to have, knowing how the development process went will be much more valuable since the scope of the project came to be a proof-of-concept and inquisitorial approach. This chapter will explain, in broad lines, how the train of thought was throughout the research and design, starting with a overview of the major milestones to depict a generic time-line. These points are in chronological order to follow the original sequence of developments, so the choices can also be understood in the corresponding context.

3.1. Overview of major milestones

To create an oversight of the complete process, a list of major milestones (points in time where an important choice was made) has been composed in chronological order. Each one of these milestones will be named and have a generic description, whereas the individual sections will contain the reason why it changed a aspect of the project. So far four of these milestones have been reached, while the last one (which is actually composed of two separate milestones) has only been partially achieved. This shortlist is as follows:

**Alan as a platform:** The biggest choice was (and is in a lot of software projects) to decide upon which language and corresponding platform to build. The decision to use Alan was based on multiple aspects, one of which was the fact that to support the whole range of Alan derivatives (and the fact that Alan defines its own grammar) using Alan would be wisest.

**Editor state model:** The first real challenge was to create a schema that could correctly represent a Alan document, while also containing all the syntactical information that can be added during lexing and parsing. Coming up with the correct structure meant that all the information needed to create or modify a document was available from a single source.

**Projects and IDE wrapper:** Splitting a project in separate single responsibility files usually increases the ease with which one can keep oversight over a project and decreases the amount of effort needed to make changes. This insight triggered the shift towards a project based IDE, since interfaces would be defined in separate files. This also meant that files would be loaded as a project into the editor, that would have to be able to edit multiple files at once, hence wrapping the editor state model in a IDE model.

**Interactivity:** Using Alan also meant using their interface, which meant binding interactive elements to certain points in the IDE model. Defining how these interactions should go, meant certain interface element types were excluded from use. On the other hand, this also meant that certain patterns would be easier to use.

**Editability and partial parsing:** To be able to use an IDE, you should be able to edit and modify the content of the files. This required faster interactions and events to modify the files contents. This constant changing in the data meant continuous parsing of the file, and since these can become fairly large, parsing a complete document every
iteration would be too slow. Therefore partial parsing was one of the milestones needed to fully complete the IDE and one of the more important choices made.

3.2. The choice for Alan

Making the initial choice of platform and language could prove to be a major mistake on the long run, since certain platform sometimes prove to lack some aspects that would enable (easier) development of functionalities. The choice for Alan was even more obfuscated because the whole platform had not fully matured yet, so not all implications on future problems and features were clear. Even though we did not know what Alan would prove to provide in the long run, the advantages of using Alan as a basis were clear: since the syntactic structure of Alan is defined in Alan itself, using Alan and including its own definition meant all Alan languages could be loaded and used without modification. This way internal references could also be resolved with no major modifications.

One of the other choices would be to hook up the parsing stack to an external editor, which would probably have to be wrapped up in a custom shell to be used as an IDE (though this was not know that it would have to be an IDE at the time). Implementing features like auto-completion and reference jumping would also need a large deal of custom code, and it was assumed that the external editor would eventually just form a front to make it look good and provide a default editable element. Although is would make certain interface based features easier, it would also mean a lot more custom code, which is why the final decision was to use Alan.

3.3. Editor state model

Using the Alan platform to develop an IDE that can handle all Alan languages requires a very solid basis, meaning there has to be a model that can represent an Alan document along with all the additional syntactical information that lexing, parsing and reference resolving could generate. Along with extra data that could be generated, the semantic structure of the document should also stay in a valid state. According to Aaby [1], syntactical information can be coupled to elements containing semantic data, and since Alan is a context free language these repetitive structures can infer additional syntactical information based on semantics [12]. The representation of the available information from these documents in this Alan structure is called the editor state model, while a instance of this model is called an editor state.

All above findings accumulated to an understanding of what this editor state model should look like and what kind of structures should be used. Initially, the document would be structured like a tree, following the structure of the document containing child elements so it would keep these dependencies and recreating the original document would only require flattening the tree. For a schematic representation see appendix H.3. Later on, this proved to make linking tokens (i.e. making a reference to an dictionary entry) intrinsically difficult because of these recursive structures. To solve this problem the model was recreated with a pseudo-tree structure, meaning that by referencing elements flat data can have tree-like properties. This also implied that tokens have mutual references to the preceding and following token (if they exist), but more important, the lines have references to preceding and following lines, but also to their parent and child lines, thus giving them all the advantages a tree has but not
the disadvantages. A schematic representation can be found in appendix H.4.

3.4. Projects and IDEs

As is the case with almost all information driven systems, the sheer amount of data available increases with time. This is by increasing number of aspects of analysis or by decreasing the amount of time between measurements. This increase in information usually results in more intricate systems to make sense of this data, which is also the case with the systems described in Alan. An increasingly complicated system requires a larger schema to describe it, and to increase readability, it has been split up into separate parts: the core schema and the interface schemas. These describe how the system produced by M-industries connect with the peripheral systems. This whole system of a set of a core and several interface schemas is called an Alan project.

This change is structural setup also needed a major change in the IDE, meaning it should be able to cope with these projects. Thus, more of an IDE like setup was created, with an interface to import, load and save these projects (importing meant reading a project definition file and creating a file that was more usable in the IDE, while loading and saving was specifically for these files) and display more than one file at a time. To do this the files had to be loaded and stored somewhere in memory, and the editor state model that was designed previously has been embedded in the IDE with a reference to the bare text file as input parameter.

This proved to be very slow, since the whole document had to be lexed, parsed and its references resolved when the document is opened in the IDE. This process was not the slowest part in itself, but especially rendering the editor state was slow with all the iterative and nested elements that required complex traversal of the rendering method.

3.5. An interactive environment

The major change described in the previous section created the possibility of introducing more interactivity whereas previously every iteration required a restart using the command line. This user unfriendliness was acceptable during the early development stages, but later on became cumbersome to test functionalities. So a graphical shell was created to make early interactive solutions possible, though the first iterations were read-only and focused on displaying documents as tabs with a IDE-like wrapper.

Additional to graphical solutions, certain interactive and configuration elements were introduced to mimic more functionalities of conventional IDEs, like tab length and autosaving to localstorage or sessionstorage (these are both webbased in-browser storage solutions, where the sessionstorage cleans itself at session end, i.e. the closing of the tab or browser, but the localstorage is persistent). These settings are also stored in the project file, so multiple people can collaborate on the same project.

3.6. Editability and partial parsing

After implementing the majority of the “simple” solutions into the GUI, what was really needed was editability. This concept is based on a state machine that looks at the current context and user input, so to visualize this idea a decision tree was made (see appendix H.10
for a simplified depiction). This tree is pretty extensive since you would want to be able to process all the different events, and so need to be declared in this decision tree. This seemed pretty straightforward initially, but soon it was discovered that more information was needed from the browser, like the current and previous caret position (the caret is the blinking element that indicates where your current position is). This meant changing the event flow from a simple straightforward one (see H.5) to one with a more complicated callback system (see H.6). This provided the tools to extract all the information that was needed to create an editable IDE, though it was never fully finished.

To optimize the parsing process in an editable document, partial parsing was examined. Partial parsing would obviate the need to reparse the whole document whenever changes are made, which is a time-consuming process. When the grammar was introduced into the editor state model, partial parsing became a possibility to implement. Since the grammar context was then available at every position in the document, it would be possible to start parsing from an arbitrary token. After thinking the process through though it was discovered that while this is possible, the validity of the document cannot be guaranteed any more after reparsing. This is because the parser has (in theory) infinite lookahead. It would require a major change in the Alan language definition in order to solve this issue and fully enable partial parsing, which was outside the scope of this project.
4. Testing

While generally seen as an annoying and cumbersome task, testing is an essential part of the development cycle. Testing makes sure the outcome of computations complies with the expected results and certain actions trigger events that (in turn) will have a specified effect of the interface, the data or both. Though testing in procedural programming and in OOP are seen as (relatively) fully developed, testing a MDD platform has not been done much (yet). This is why some assumptions on testing had to be made.

4.1. Testing in Model Driven Development

By using a model driven development approach, writing software is very different than when developing in an object-oriented way. This naturally also requires a different approach to software testing. It is customary in object-oriented programming to model the system that is going to be built with class-diagrams and other UML tools. The classes are then implemented with the necessary methods, interfaces, etc., in compliance with the model. Methods and functions are modularized so that a single method or function performs only a single functionality, to make the program less ambiguous and allow for better unit testing. Unit tests can be seen to have the primary function of enforcing the code to comply with the model design.

In model driven development the model of the system is not a set of diagrams, but a well-defined structure that is instantiated in the actual program. When a structural mistake is made in writing the model, the platform will report an error. If the model is structurally correct however, it can be parsed and an API can be generated for it. This API is comparable to the implemented classes of an OOP project. The result of this is that unit testing is not necessary any more, because if there was a bug in the generated API, it would be a bug in the platform and not in the system that is being developed. This obviously leads to a lot less necessary testing.

The software that is written using the API can still contain bugs however, but the API will limit this in a couple of ways, if used correctly.

Firstly, it gives type-safety. In the JavaScript API this is less strict because of the dynamic nature of JavaScript, so it is still necessary to pay attention to the types and give appropriate names to variables, but if a mistake is made the program will fail if the wrong type is used. This problem can also be countered by asserting the type of the arguments at the beginning of a function.

Secondly, because the API is generated around the model, it also follows the structure of the model and so the program will not deviate from it.

Finally, also because of the previous point, the API enforces the programmer to implement all cases of the model logic. For example, say there are three types of 'user' defined in the model: administrator, system user and client. In the program that is using the model, we might want...
to iterate over all the users and perform some action dependant on the type of user. Then,
we either have to write an implementation for each type of user and no other type, or, if we
are interested in just a single type, we have to implement a case for the preferred type and
for when it is not the preferred type. This way we are forced to consider all cases at all times.

Still, the code that uses the API needs to be tested. The way to do this, is to make test
instances of the model that when taken together cover all the aspects of the model. For
example, if we test the model in the example above, we would make a test model instance or
test models instances that have users of all the three types, viz.: administrator, system user
and client, and then check if the behaviour of the program is correct. In the next sections we
describe how we tested the different parts of the system and where we had to deviate from
the testing scheme described here.

4.2. Testing the lexer

There is no model for the lexer, so right away the testing of this part is not completely as
described above, but still not far off. Because the lexer is in essence a finite-state machine, so
in order to test the lexer, it would have to be brought in every state and make every possible
transition between states. This can be done very easy by lexing a piece of text that has the
correct structure to terminate in a state. If handled correctly, the lexer should either insert
an error and the type of error, or wrap up the lexing in a correct fashion. This choice is made
by looking at the type or error made and how breaking it is for the structure of the document.

Most of the lexer testing can be done by trying to lex various text structures and com-
pare the result with an expected document representation. The only thing to be tested by
plugging into the internals is to follow the states it traverses while lexing text.

4.3. Testing the parser

The parser, as opposed to the lexer, is based on a model. This model is (based on) the
grammar model. So it would be necessary to test the parser with a grammar instance that
incorporates every aspect of the grammar model. This is not so much of a problem, because
(almost) all of the languages the company uses have grammars that do this, including the
grammar itself. In the earlier versions of the parser, it would receive two input structures:
the grammar instance, which is part of the language definition, and the sequence of tokens
received from the lexer that are to be parsed into a raw language instance of which the gram-
mar is part of the definition. Both of these combined determine which parts of the parser are
executed. The revised parser that directly updates the editor instance had only the editor
instance as input variable, which had the grammar already integrated.

We tested the parser by running it with a valid document that was as small as possible
that would run all the functions of the parser. By examining where the parser fails, we could
determine where the error originated. If the document parses without exceptions, the test
passes. Next up we created a set of copies of the test document all with distinct alterations
that should trigger parsing errors, to see if the parser returns an error when it should. With
all these test documents, the full behavior of the parser could be tested.
4.4. Testing the reference resolver

The reference resolver follows, like the parser, a model and a secondary data structure. In this case that would be a reference path defined in a language schema and an editor instance. Both of these structures are verified and the code of the reference resolver relies heavily on the generated API’s of them. Testing the reference resolver then requires executing all the parts and verifying that no exceptions arise. To do this, we build test documents with valid references and invalid references so that every part of the reference resolver is executed.

4.5. Testing the user interface

Interface testing is one of the most exasperating aspects of testing. This is due to fact that the way you can test it is usually bound to factors like interface platform, resolution and render engine. Though headless testing (testing the interface by mocking the rendering part, so you will not have an actual interface) is becoming increasingly popular, it has not reached its full potential. Several webbased headless testing suites are readily available, but all of them rely on JavaScript to run their tests (like selenium[10]). Other platforms can test more standard interface technologies (like JavaFX[4] or Qt[2]) but are more cumbersome to create fully fledged test environments for.

Because most of the interface tests follow the “if I click here, this should happen” method, in the new platform you can test the interaction with interface elements by firing the correct event bound to a mocked element, making the test suite reactive and dynamic. This is possible now since the behavioural and aesthetic definitions are now bound together, but are separated from all other elements. All interface elements during the project phase were described in looks only in one big render engine, and event were statically linked to event handles.

4.6. Performance testing

When most people think about performance testing, speed is the first thing that comes into mind. True, this is the most superficial measurement of performance, but even something simple as speed is divided into several aspects, like reactivity (how fast is an event passed on to the correct event handler), processing time (how long does it take to respond from an event) and update time (how much time does it cost to respond to the response and update the interface). This is a good indication that performance testing is not that simple. Also, acceptable delays are subject to interpretation that, in turn, are subject to many factors like age. Older people are usually less annoyed by a bit slower reaction time, and developers on the contrary crave speed to evaluate the results of their actions.

Another widespread measurement of performance is memory usage. As said in section 1.2.2, memory could become an issue since the whole document and its representation within our IDE will be loaded fully in memory. Whether this is a flaw or not can be decided by evaluating the value of having (almost) no server calls against the value of having all data and information available at all times. Since Alan is a context free language (see section mindustries alan language reference here) and the document representation model followed that structure (and thus is also context free), the memory usage increases linearly with document size with a fixed startup amount.
Measuring speeds is easy with simple developer tools and, in the case of web development, these are readily available inside the browser self. Though we were constantly busy with performance, we did not have the time to create an extensive test set and monitor the performance. These intermediate results can be found in appendices H.11 and H.12.
Conclusion

The objective of the project was to create an IDE for M-industries that supports their in-house developed languages, especially the Application language and Alan projects. The most valued functionalities of the IDE are the ones that existing editors could not provide. We delivered a working IDE with support for Alan projects and powerful language specific features. This was achieved by creating an editor state model that is linked to the language definition models. This way the editor can give powerful context aware and language specific information, while not being language dependant. If for example the Application language would be modified, the IDE would not have to be modified. Below is a list of the features that have been accomplished.

**Alan project support:** Alan projects are what the company define client applications with. To support Alan projects an IDE wrapper was build around the editor state model, so that multiple editor states could be initialised for all the project files.

**Loading and saving projects, with user settings:** Along with the IDE wrapper, support was added to save the entire IDE state, along with user settings, and load it again at a later time.

**Support for every Alan language:** By linking the Alan language definition models to the editor state model it is possible to load any language defined in Alan into the editor.

**Code folding:** By counting the indentation of each line during lexing the lines are organized in a tree like structure to enable code folding.

**Error message display with suggestions:** The IDE shows errors in the file that is worked on in an error message field below the editor window. These error messages show suggestions when appropriate which can be selected and then inserted in the file.

**Language dependant syntax highlighting:** The IDE supports syntax highlighting by means of a complex token structure in the editor state model to which different styles are coupled. Because the editor state model is not defined at language level but at meta-language level, the syntax highlighting will differ for every language, dependant on the language grammar.

**Jump-to-Definition functionality:** References are a key component of Alan languages. By implementing a custom reference resolver for the IDE, the references could be linked to their definition enabling the jump-to-definition functionality.

**Auto-completion and suggestions:** Because the editor state model is linked to the language definition models, the editor can give smart suggestions for valid options as the user is typing.

There are features that the IDE lacks. For example, text is not fully editable. However, the most important objective for the IDE was to support features specific for their in-house languages that other editors do not, and that objective has succeeded. There also Alan language specific features that have not been implemented due to time constraints, but the current IDE is a solid basis to which these features could easily be added.
Discussion and Recommendations

The project was a big challenge. Building an IDE alone would have been difficult enough, but developing it in a whole new programming paradigm was a step beyond. The model driven approach of M-industries is a long way apart from the object-oriented way of programming that has been taught during our bachelors. We were thrown in the deep end of the pool, and it took a long time to adjust to this new way of developing and even at the end of the project we were still not completely confident in MDD. There were times during the project that we no idea of what we were doing, but once we settled in it became difficult to go back. To deliver a working system that is complex as it is, developed for and with a whole new programming paradigm, we can be rightfully proud of ourselves.

Nonetheless, there are features that have not been completed or just not implemented due to time limitations. We leave them as recommendations for future work. Below is a list of all the features that did not make it.

**Jump-to-Meta-Definition**  While Auto-completion and Suggestions are already very useful features that allow insight into the grammar of the language, it might sometimes not be enough. In that case a jump-to-meta-definition feature could be the answer. This feature could be realised either by opening a new editor window with the grammar loaded in and the appropriate line highlighted, or a popup floating above the editor window with several lines from the appropriate place in the grammar. The former solution would probably the easiest and most versatile. It would not be difficult or require a lot of work to add an editor instance with the grammar into the IDE, but the link from the grammar instance and the grammar itself could lead to difficulties.

**Reference Resolving between Project Files**  Alan projects were introduced by the company somewhere halfway through the project. This required not only a system overhaul by us, but also in the tools from the company itself. Reference resolving was a difficult part of the project and resolving them within a single file was in itself already a lot of work. To be able to resolve references within different editors in the IDE, there had to some modifications to the Alan implementation from the company, which would have been doable but because of time concerns we decided not to implement it.

**Suggestions with Reference Resolving**  This feature was already suggested in section 2.8, suggestions could be expanded to also include reference information. Currently the suggestions only encompass information from the grammar, but since the language grammar and schema are linked in the editor state model it would be possible to also resolve the potential references (if possible) and show suggestions for those too.

**Full Editability**  Editability never had high priority, because contextual language specific information was valued more highly by the client. Nonetheless, if the document can be analysed and edited in the same program that would be an obvious asset. The IDE does have limited support for editability and is not far from having full editability, it would just require more work.
Bibliography

A. M-industries

In the year 2000, a new concept of system development formed in the mind of Corno Schraverus, and signaled the beginning of a decade of searching for the perfect implementation of his idea: A language workbench based on MDD technologies to enable domain specific languages to be created on the fly. Eight years later he found his first partner, the first person to whom he could explain this concept and have him understand it. With him the shift from a language workbench for language designers to a web based application system for solutions. Not long after that, another person entered the company and their first client became reality, thus the company called M-industries was born [6]. This concept consisted of using a more generic approach to MDD for systems development, using the models to define the system behaviour, while “running” the model in a framework that could generate a default interface consistent with this model (the first steps in UI generation were taken in 2010, creating a complete software solution platform). This meant a huge boost in rapid prototyping because you could edit your model and generate a new API and interface, without having to think about complicated internal mechanics. This was their vision then, and with the aid of the TU Delft and its students, this has become reality now.

While the first modeling systems and languages were archaic and imperfect, they signaled a start towards the implementation of a new programming paradigm. Since then these have been improved along with the core elements of the company has changed and the focus is now on platform improvement and system implementation. Though this does not mean that the platform is not finished yet. Internally, there are two methods: - A C++ approach, with a compiled datastore and hardware access, - and a JavaScript approach with a more dynamic framework, specifically designed to enable fancy interface implementations and custom events. While both approaches provide the same internal functionalities, the eventual usability does differ, since the C++ framework is faster and has a few features that are not found in the JavaScript framework and vice versa.

Since the founding of M-industries, platform development has gone really quick, and eventually students from the university came in to do projects. Some of these stayed with the company, continuing work on their project, or improving the platform(s). Since it’s a regular language [12], other, more complicated, languages can be designed with Alan. This provided them with the tools to define multiple use-specific languages to make development a lot easier while still providing a complete software solution platform. An introduction to Alan can be found in Appendix D.

The ultimate goal is to start a new software revolutions like object oriented programming did between 1980 and 1990 (see Appendix (C) for a more elaborate explanation), making programming a more data driven business by shifting from procedural programming methods, and making software development a more accessible trade. This new shift will make building software more like drawing a blueprint: all the structural elements are defined and all possible states are defined in one model, but how the mechanics are used can be designed afterwards. Imagine a black box where you can put a blueprint for a building in, it will build the structure, but the material and color of the wall are yours to decide afterwards. Also, the choice between using a push-to-open door or one controlled by a button is complete to the user. This black
box is the platform M-industries built, and the blueprints are drawn in the language Alan. By providing hooks to control the mechanics, but also by removing the requirements of implementation and validation, the developer has a huge amount of freedom, while being free of the standard programming problems concerning the internals.
B. Original project description

At M-Industries, we use a set of custom domain-specific languages covering different aspects of application development to build solutions for industrial multinationals like Tata Steel and Sapa. Some examples of these languages: - Application model (to specify entities, relations, derived properties) - Widget markup (our browser-based UI engine renders these widgets) - Data migration (e.g. to migrate data between different application model versions) Existing editors and IDEs like Visual Studio, Eclipse and WebStorm have no support for syntax highlighting, code folding, autocompletion and error checking in these languages. We are moving towards intensively using these custom languages in our own tooling and we would also like to make them available for consultants and end users. To do this, we need a modern editing environment with support for features like syntax highlighting, code folding, autocompletion and error checking. We would like you to design and develop a browser-based code editor that supports our set of custom languages, including the features mentioned above. You can start your project with an analysis of the available options, comparing existing editors like CodeMirror and Cloud9’s Ace. The project will also involve working with our parser engine, performing static analysis on parse trees and dealing with parsing incomplete input data (partial parsing).

B.1. Company description

M-Industries is a startup company from Delft, developing software for clients in the industrial sector. It focuses on designing systems that support complex business processes by using its in-house developed data-modeling language and software development platform, based on several years of research and development. The company has special interests in domain-specific declarative languages, functional programming, data transformation, static code analysis, code generation, browser-based applications and Node.js.

B.2. Auxiliary information

Our office is located at the edge of the TU Delft campus, near the Delft-Zuid train station. We have workplaces available, and we would prefer it if you perform your work at our office. Free lunches are available. Students are compensated with a monthly stipend. We can support Dutch- and English-speaking students.
C. Model Driven Development

MDD is a relatively new programming paradigm \cite{11}\cite{13}\cite{8}\cite{9}, with both pros and cons. The concept is based on making the data less abstract than a Object Oriented system, which in turn has a more concrete data form than Procedural Programming methods. These methods use the algorithms to give the data shape, so the information is subject to the algorithmic properties of the system. Languages like these (C, Fortran or Pascal) usually are fast and need very little to no platform to run on. This does also imply that the validity of the data is not ensured through the algorithms. This data is hard to interpret for humans, since only inherent and underlying structures can be used, making debugging and testing a tough job. Since the data structure was not immediately apparent, programming was not only about writing code, but also a lot about resource and process management.

Since the amount of data in a system is ever increasing, this data should be organized in a more natural way. This is where Object Oriented Programming (henceforth OOP) is used, in a large variety of languages like Java, Python and Ruby. Since the models in OOP represent an abstraction of the information contained in the system (or more concrete in the procedural programming perspective), and the algorithms are designed to work with and manipulate these models, data validity is ensured. Complex system become a lot larger though, since these algorithms have to use more steps to perform data manipulation. These models have their own methods that can access and manipulate internal data and data-structures, but can also be easily extended. This makes OOP a more accessible type of programming for a modular or scalable system.

Reading and writing these models and algorithms can be very cumbersome, and is overly complicated for most system. MDD is the paradigm that is based on the thought that the form of the data should dictate the possible states, and with that the algorithmics in a system. If the model is correct, the system can only be in a valid state, also meaning that the information contained in the system always has a valid form. Debugging and extending is very easy, since a modification to the model is a modification to the system, and generating a simple interface bound to this model can increase the speed and ease with which one can modify or verify the system. This also relieves the developer of the need to create interfaces or communication channels since the system will generate an API (Application Programming Interface, the hooks of a system which external systems can use) to be used for data manipulation and data extraction. Imagine a machine which you only need to feed the blueprint to your house and it will build the skeletal structure, floors, walls and roof. When the system is done generating and its running, all you have to do is paint the walls.
D. Platform

In this appendix we describe the platform on which Alan is built and thus the platform on which and with which we build the IDE. We describe how an Alan language is defined, which is an important and central part of the editor state we developed. We will further elaborate what Alan projects encompass, which is an important feature for the IDE. Finally some of the platform tools are described.

D.1. Language Definition

A language definition consists of two parts: a grammar and a schema. The grammar describes the syntax of a language and the schema describes the semantics of it. Both of these are used to convert a document containing raw text into a program or, more accurately, a model. The grammar defines the structure a program or model can have in the language. It contains specific rules about the order in which words and symbols may occur in the document, and what type of nodes and properties the model will contain, depending on that order. It also contains a list of keywords that can occur in a language instance. The schema in turn defines the meaning of the nodes and properties that can occur in the model. While the grammar denotes that a node or property exists, the schema describes what its meaning is in the model, or what relation it has to other parts of the model. Such a relation is called a reference.

Alan can be seen as a base language in which (in theory) infinite amount of new languages could be designed. This due to the context free structure Alan has, meaning that there is a base grammar that is very simple providing the tool to create more complex language structures. Alan could be seen as a box of very basic Lego: blocks, sheets, pivots and slopes for instance. Using these atomic pieces complex models can be designed that could be shrunken into standard size blocks to be used in other models. Due to the context free-ness, creating a recursive structure is also possible, enabling you to model very interesting systems.

One of the most useful pieces within Alan is the use of a reference. Since Alan relies completely on the structure of the model to dictate the elemental mechanics of the system, a reference can refer to one specific place in the model only. This also makes sense since using references means all the information that belongs together is put together in a block, making finding all users in a system a superfluous operation since they can only be in one place. References can also make conditional steps, for instance: a certain block can only be used with a female employee, meaning the employee reference could have a conditional step to ensure all references made are to a female employee, thus making the information intrinsically correct.

D.2. Projects and systems

At the end of the project, increasingly complicated systems were being designed. Up to this point, all interfaces with external systems and internal mechanics were modelled in one file. This did not improve the readability and debugging. To counter this, a model of a project was created, that was written in a domain specific language created from Alan, along with
several other languages that would create a complete developing package. By splitting the application model (the core model), the interfaces (the models with which external systems could communicate) and the links between these interfaces and the application model, all with their own specific languages, creating a system with external systems has become more easy, faster and modular. This meant that other systems that connected with the same external systems could re-use these interfaces.

These projects are defined in a file, which modelled the file structure for this project, and the links between these files. This model is extensible and is readably by the IDE to be able to cope with varying project setups. This also meant that the IDE has to be able to solve and link references between these files, while not being in the same document instance.

D.3. Platform Tools

In this section the platform tools that have played a role in the design of the IDE are listed. The fabric server is the server where the IDE runs on, the datastore is a tool that we have considered using but opted not to and the language tools have been modified specifically for the IDE as is described in Chapter 2.

D.3.1. Fabric Server

This is the base platform for all systems built in with the Alan language. Since the platform is web-based, a server is started that handles all requests and manages the data on the front-end. This also is the platform that generates the interface using the defined GUI-interfaces and event handles.

D.3.2. Datastore

For applications that need data persistence, this provides a database that can be compiled using the specified model so the internal structure is optimized for the current system. The datastore is build specifically for the Application language. For editability of code in the IDE we have considered writing the project in the Application language to make use of the model mutation functionalities of the datastore, but the Application language proved to be unsuitable for our project.

D.3.3. Language Tools

The following set of tools are necessary to be able to work with an Alan language. As described above, a language is defined in a grammar and a schema. The parser parses a raw text document into a ”raw language instance”, using the grammar as a syntax description. The API generator in turn generates code around the JSON structure, decorating it with methods for walking the structure. In this process all the references defined in the language schema are resolved.

Parser

The job of the parser is to convert a raw document, a series of characters, into a raw language instance. This is a JSON structure made according to set rules, which complies with the grammar. The first step of the parsing process is lexing. The lexer preprocesses the
The list of tokens produced by the lexer is then passed on to the parser. The parser then traverses the grammar, consuming tokens one by one. If it expects a certain keyword, for example, it will check if the first following token in the list of tokens is that keyword and if it is, it will consume that keyword. If the first following token is not the expected token however, the parser will quit and return an error message stating that it failed, why it failed, and what token it did expect. It can also happen that the parser has to make a choice which path it has to take in the grammar. In fact, this will always happen, otherwise every program or model in the language will be the same. In such a situation the parser will look at the first following token, and check if it is a valid option. If so, then either the decision can be made, or another token has to be considered in the same manner. If the token is not valid, the parser will once again return an error. These decisions are not about what token to consume, but rather what type of node or property there should be in the raw language instance it is creating. Every option in a choice must be distinguishable by a unique combination of tokens, otherwise the system would obviously not be deterministic. The parser actually does not create only one raw language instance, but also a shadow copy with locations added to the nodes. The parser keeps track of the location that was added by the lexer of the last consumed token. When a node starts, this location is added to this shadow copy. The shadow copy will only be used to look up locations in case of an error during the generation of the API. The original raw language instance will be used to create the API and the final language instance.

API Generator

After parsing it is certain that the structure of the model is valid, but it is still not certain if it is a valid language instance. For the document to be a valid language instance, it has to confirm to the schema. In this definition, the references between the model entities, e.g. nodes and properties, are defined. These references, are defined in the schema as paths from one property or node to another, either directly or indirectly. These paths are followed in the language instance. If the reference is valid, a pointer to the end-node is stored at the referring node, and a pointer to the referring node is stored at the end-node. If the reference can not be resolved, an error has to be thrown with information about the location in the document where the referencing node is, and what path it tried to resolve. The latter is easy, because it can be directly copied from the schema. The location in the document however, needs to be retrieved from the shadow copy of the raw language instance that was made during parsing. To do this, a path to the node in the language instance is build, and then this path is tracked in the shadow copy to find the location in the original document. During the resolving of all the references, a comprehensive API is build around the language instance. This makes the model read- and writeable and applications can be build with it. This might be the goal of this step, but it is also currently the only way to validate the model to the schema.
E. Preliminary research report

Final Bachelor Project Research: A webbased editor for m-industries
Elgar de Groot

Stijn van Schooten May 5, 2014
Introduction

Working with your own custom programming language has its perks, as well as its difficulties. One of these perks is that you can design the language as such that it grasps at the core of the problems that it will solve or describe. But having designed it yourself, no editor will support you language. This research paper will describe the pre-emptive research we have done on the roadmap to building a webbased editor with support for specific custom languages and a custom featureset, as is our assignment for m-industries[2].

Before starting on a project, you need to know what the core of the project is, so we will start the paper with a chapter defining the project in detail. Hereafter the scope and the shift in scope of the project will be made clear, along with the context of the current scope. Here, the division into the two solution domains will be discussed and the interface through which these two domains connect (which is the core of the problem) also will be made clear. Next will be a chapter on the features that are desired, whether this set of features is achievable is not clear, nor important. Each features importance, functioning and requirements will be explained, and the current implementation path will also be described. A chapter on the issue of visualising the editor will follow, listing some of the challenges. The final chapter is a conclusion containing a short recap of this papers content, followed up by a intermediate conclusion to the question: “what do we need for the editors backend?”. 
E.1. Project description

A very well known proverb between developers is “It’s better to make a good copy than have a bad implementation.”. This means that using another developer’s idea in your system is common practice (it is actually impossible code without re-using some code from someone else), but also that it is better to try and not start from scratch when possible. This is not the issue for our system, since the mapping between the systems will never correctly reflect either of the systems, and makes for an ungly mapping that is bound to be slow. This is the current insight for this project, which meant a major shift in project description, which shift will be depicted in the first section by first describing the origin project description, after which the differences with the current description will be detailed.

E.1.1. The original project

Originally, the assumed task was to build a webbased editor that supports the custom languages of m-industries. This would (probably) mean taking an existing webbased editor and modify it to fit their needs. Using an existing editor would eliminate the tasks of visualisation and handling input. If this would have been the case, one of the first things to do was integrating their language grammars into the editor, to support syntax highlighting. This would require hooking m-industries’ own lexer and parser into the editor and using that output. This output could also be integrated with other features to create a functional editor, since the parsing tree contains all semantic information that can be extracted from a document.

This classic programming approach would lead to a feature rich editor, that does not comply with the desired methods of the company, but more information on the context of this project in section E.2.1. This approach means that the project would consist of building a list features into an editor, using the API of their own system, without the need to design a intrinsically functional system on the backend. Working down a list of required features is the most common way to develop a system, usually with building the framework, on which the system will run, as a first feature.

Agile and modular development methods would ensure that a change in the feature list would not result in discarding the current code-base. Thought this does not mean that the development path is the correct one, but it does make it easier to modify the direction you are headed more easily. Using an existing editor as base point, the requirements per feature are easier to define, since they have to fit the editor’s API. The more these requirements overlap, the shorter the implementation time would be for that set of features, since a good modular programming method would reuse these overlaps for multiple features.
E.1.2. Shifting of the project

After almost two weeks and a lot of discussions the project was redefined, taking a completely different approach and having a more concrete target. The actual editor functionality has become a secondary target, shifting the main focus of the project to a backend for the editor features that is integrated in m-industries’ system. This means that the mapping between the systems does not exist, leading to a more consistent system. This new method forces you to meticulously define the core of the problem, making it possible to create a whole system with only one definition.

Since this schema definition generates a graph database that listens to updates, the modelling of the editor this way could greatly benefit from functionalities like partial parsing and partial re-rendering. This would not only improve the performance and responsiveness of the system, but would also shorten the paths the data has to travel, making the system much simpler (after a more complex design period).

The focus came to be more on having a solid backend for the desired features that has an API that could be used to hook into the editor. In chapter E.4 the situation with the visual frontend will be explained more, that is of secondary importance because visualising the editor in bound to design and layout decisions, which would distract from building a functional feature. While in the previous understanding of the project, (basic) visualisation was one of the first milestones we wanted to tackle, since the results of the backend logic could be visualised.

Also, the use of conventional programming methods and strategies were undesirable, as one of the requirements became that the logic of the backend would have to use their own model driven system (see chapter E.2.1) to enable m-industries to extend the editor more easily and integrate the editor into their own software package. This meant taking a completely new approach on the problems and solution methods, since the model and the state of the model now were the basis on which the system should run. This also forms the main challenge of the current project: how to describe the logic behind an editor (at least for the defined set of features), in m-industries’ model driven design methods, i.e. in their schema language.

Creating a system by modelling it in a schema can also be done with agile methods. Since the components and the state in which these components could be in are described, you could start with a more basic version containing less features, while still being completely functional. The modular aspect in modern programming methods actually is built in to their schema language, since reusing a model component can be done by defining a separate component in the schema, also making recursion possible.

All in all, there has been a major shift in approach (from common programming methods to model driven development in their own system) and target (from building a functioning editor to a intrinsically correct and functional backend) for this project, making the first two weeks very turbulent. In conversations with Corno Schraverus (the CEO), their unique look on problems and problem solving have left us with heads spinning from information overflow.
E.2. Project scope

In order to clearly describe what sort of solutions to the project are possible we first outline the scope of the project. Since our client is M-industries we will first give some background on the company, the way they develop software and their technologies. Some elaboration of the languages for which the editor is meant is given. In this context, we will describe the two solution domains in which we believe the solution can be divided.

E.2.1. Context: m-industries and the schema language

M-industries use model driven software development as their design philosophy. This means that in order to tackle a problem, one would first create a complete model of it. Such a model is written in their own modelling language, the 'schema language'. A supporting framework can interpret these schema’s and generate an application from it. Consequently, a model written in the schema language does not only describe a system, but also determines the implementation of the system. This schema language contains the bare-bone building blocks to model everything. This language defines itself using a grammar. It is possible to create a new DSL by writing a grammar and a schema defining it, using the schema language. In this way they have defined a superset of the schema language, called the application language, which they use to write the actual applications for their customers. This will be the main language for which we will implement the editor features, but the self-defining nature of their languages will most probably result in that when a solution is found for one language, the solution for the other languages will be a trivial step. This statement will have to prove itself, so we will first focus on just the application language. Because the application language is a superset of the schema language, the building blocks are the same and thus the most important aspects. We will give a short description of the schema language below to aid in understanding the context of our assignment.

Schema language  A model written in the schema language is actually a big tree structure of uniquely named fragments with uniquely named properties. A property can be a one of several pre-defined types. For example a dictionary, a list, a sparse or dense matrix, a group, a stategroup or a reference. There are three types of tokens that can be distinguished: keywords, whose semantic meaning is specified in the grammar; literals, which can either be a key in a dictionary or matrix, a reference or just a text; and numbers, which are numbers. For a more detailed description of the schema language we would like to refer to the thesis of Joost-Wim Boekesteijn[1].

E.2.2. Solution Domains

For our assignment to create a language sensitive editor we discovered that for every feature we have to find a solution in two domains. On the one hand we need to gather information on the code, may it be syntactical, grammatical or semantic information. On the other hand we need to represent this information somehow in a graphical interface. We’ll refer to these two domains as backend and frontend, respectively.

Backend  The information we can get from the backend is the lexer output, parser output and, trivially, the plain text of the code. We have access to the lexer and parser-generator code and are free to modify them to our needs, if we need more information or different output from them. The current lexer is written in JavaScript and returns a series of tokens in
JSON format. As described above, these tokens are either a keyword, a literal or a number. The parser is also written in JavaScript, and it too generates JSON output. It needs a parse strategy that is generated from the grammar by the parser generator (which is actually a parse strategy generator).

**Frontend** At the frontend there must be a rendering engine that can show text and handle input. It should have hooks for all the data from the backend, and hooks for the backend to propagate user input. It’s a representation of the backend, with capabilities of indirectly altering the backend. More specific information can be found in chapter E.4.
E.3. A list of features

Every feature in the following list will describe a few things like the basic functionality, possible variations on the feature, the required information for that feature to work and a description on how that information should be used. Please do keep in mind that this is only a projection and some features might be added or removed.

E.3.1. Syntax highlighting

Reading code can be very difficult, even when it is your own code. Syntax highlighting is a very useful tool to make code more readable, by colouring the different types of tokens with their unique colors. This way you can see at first glance what is a keyword or what is a literal, giving you greater scanning speed.

The token tree will be used to find what kind of token that token is, giving it a specific colour as has been defined in the configuration.

E.3.2. Code folding

Navigating the code can sometimes be very cumbersome due to long nested lines of code, forcing you to scan every line until you find the one you were looking for. Folding blocks of nested code will make this much simpler. You will still see the defining line of parent code, so you can still navigate, but the nested lines are invisible so you can find your target faster. Atomic lines (lines with no children) will not fold, as is the case for empty lines. The folding range can be found by inspecting the range of lines of the children, since they will contain the range of their children and so forth until a line has no children.

For this to work in the easiest way, the editor should at least have access to a token tree, since a node contains a line (a series of tokens) and has a list of children nodes. Toggling the folded state of a node should also toggle the visibility of the children, without modifying their states.

E.3.3. Code completion

Most programmers will admit that the most useful tool an editor or IDE can have is code completion or code suggesting. What the editor essentially does is see what can possibly come after the previous token according to the grammar of the language. All these possibilities are listed in a visual way, but should also enable the user to select a suggestion that will be entered. Filtering this list can be as simple as typing the first letters of the word you are looking for. A more advanced version could also give placeholders for the arguments needed, along with the correct structure to make it even easier.

Knowing what can be entered here is knowing in what parsing state you are at the moment, and knowing what kind of references can be made. So for this feature the most important requirement is to have the parser tree and reference list. This information is bound to the tokens in the token tree, but that should already be available since that is a requirement for parsing and resolving the references. The editor will have to be able to navigate to the correct parsing state according to the current cursor location. There will be a (set of) choice(s) possible here according to the grammar. These choices can be resolved with the references, which produces a set of suggestions. These are fed back in the editor to be displayed and
possibly be filtered. While not a requirement for the core functioning of this feature, partial
parsing will make with a lot easier and can give the running time for code completion a
boost.

E.3.4. Jump to (meta-) definition

Writing code or models is very difficult if you have no references to use, or it might slow
you down significantly when these references are not easily accessible. Imagine this situation:
you need some help remembering the parameters of a specific component. You hold down
the modifier key and click the component. Instantly you are being navigated to the location
where that component was defined, providing you with the information you needed. An al-
teration on this function might be not actually navigating to the definition, but showing the
definition in a popup or in a area specifically for this purpose.

If you analyze this behaviour, some requirements come up: you need to know the location
of the reference target, meaning you need to resolve the identifier against the definitions in
the parser tree, i.e. you need the parser output. Another requirement is the need to know
what kind of referral has to be resolved, since a reference to the grammar or the language
definition can also be used.

This feature could speed up the process greatly, since looking up references externally kind
of is a pain in the ass. This does also imply that the editor in use is aware of the language
used in the current document and has readable information on this language. To increase the
usefulness of this function even more, a more readable format could be put over the definition
information.

E.3.5. Context resolving

Model driven development has strict context rules, you can not refer to something in a higher
context than you are in at the moment. This can be solved by making an explicit reference,
using context identifiers to verify the context you are escaping from. When making such a
reference, resolving the type of context now has to be done by hand. A feature that highlights
the context in question (the currently reference context or the context belonging to the se-
lected identifier) can be of much use, making references much simpler to write. But since the
editor now knows what context it is in at the moment, it can also automatically supplement
the context escaping identifier with the correct context symbol.

To be able to make these references, the parser tree has to be resolved and the context
evaluated. This context can be used to find the correct context escape symbol and find its
parent context. These could also be used fairly easy to highlight the current context.

E.3.6. Editable text

What is an editor without editable text? Just a static representation, a picture, if you will.
This is why every input event in the editor should lead to an update of the editor state. A
mutation of the text should be stored in the state, triggering a re-rendering of the accompany-
ing line, letting you see the mutation. How this storage and updating is done efficiently is not
clear yet, since different editors use different methods, and academic research suggests even
more different methods. But the mutation mechanic will always be the same: on specific small
updates only the value of the token will be mutated in the token tree, otherwise the token
tree itself should be updated, possibly triggering an update event further down the chain.

Making text editable will require a consistent and representative editor state that also contains the current context information, like cursor position or selection range.
E.4. Editor visualisation

Editing code (or any form of text) is done on a graphical basis, the words are graphically represented and you have cursor(s) where you can edit the content. But for an editor to be functional, the representation is not at the core of the problem. Though an editor without a graphical representation feels useless, in essence it is not. In our case, the actual visualisation of the features can be done very generically. Because it is a web based system, standard css and html methods work, giving a very large range of freedom. If our description of the editor is in essence correct, the logic will also be correct. This correctness will also translate to the frontend, meaning that creating the graphical shell only requires using the information structure to generate a html representation with a appealing styling.

The model driven methods give the frontend developer the freedom to create a mapping between the backend data model and an enriched/abstracted frontend data model. This frontend model should contain enough information to enable the engine to render this information in a sensible way using predefined GUI elements (called widgets), creating, in essence, a new widget specifically for your model. What rests is styling this new widget, and hooking it up to the system. This way the widget is very close to what the actual information on the backend actually represents.

This widget is defined in a language that is derived from the schema language (see section E.2.1), that is able to describe how the data model that is fed to the engine is used to bind the components of this model to widgets. This method means that deciding what to re-render and when is not necessary any more, since only the component bound to the changed information is re-rendered on the fly. For instance: if you would represent a document as a series of lines, the modification of a line would invoke an update event on the widget rendering that line, only re-rendering that line.

Instead of rendering the document from scratch, a part of an existing editor could be used, to only do the graphical representation and handling input events (like keyboard or mouse input). If this would be done, a mapping between the two systems would update the state of the backend for every event on the background, and a change in the backend would have to fire a re-rendering event. This would take care of a lot of styling issues and would mean that we would have to make a model that described the logic behind the editor and not the editor with its behaviour, but we do know if that will work properly or if it will fit in the system. We are also not sure yet if the document representation on the front and backend should be mapped, or that it should be updated from the raw text for every update since the difference here could be significant. But this is a problem that will have to be solved when working on the graphical representation.
Conclusion

The main objective for our project will be to find the best way of linking the language specific information to a graphical interface. The approach will be in twofold. For the frontend, we must create a model in the schema language of m-industries that on one hand represents the possible editor states and on the other hand makes editing source code and displaying semantic information about it feasible. In the backend, we must change/rewrite the lexical analyser and the parser so that they return the desired data structures for our editor. For the time being we will primarily focus on the backend part, since these provide the actual functionalities.
Bibliography


F. SIG feedback

F.1. First evaluation

De code van het systeem scoort 2 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code ondergemiddeld onderhoudbaar is. Een hogere score is niet behaald door lagere deelscores voor Duplication, Unit Size, en Unit Interfacing.

Het eerste dat bij het bekijken van de code opvalt is de naamgeving. Jullie gebruiken "lib" als naam voor een component. Voor ons was die naam enigszins verwarrend, aangezien deze naam meestal wordt gebruikt om third party library code van de "eigen" code te scheiden. Aangezien deze bestanden geen copyright-statements of andere headers bevatten gaan we er van uit dat het hier om door jullie geschreven code gaat. Als dit inderdaad zo blijkt te zijn is het aan te raden om een andere naam te kiezen, aangezien dit een veel gebruikte naamgevingsconventie is en je zo verwarring voorkomt.

Voor Duplication wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren. In jullie code komt dit met name voor in het bestand editorApi.js. Een voorbeeld is het blok code met callbacks voor "yes" en "no", wat in regel 1721-1749 voorkomt en onmiddellijk daarna in regel 1752-1780 wordt herhaald, en daarna in regel 1783-1810 nog een keer. Aanpassingen in dat stuk code moeten nu drie keer worden doorgevoerd. Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de callback die jullie aan module.exports meegeven in het bestand fillProject.js, zijn aparte stukken functionaliteit te vinden welke gerefactoerd kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld ’// Add directory entries’ en ’// Update folder entry references’ zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Voorbeelden van methodes in jullie code die erg veel parameters hebben zijn processLoop en processInstruction in het bestand parse.js, en MissingEditorFunctions.walk in het bestand missingEditorFunctions.js.

Als laatste nog de opmerking dat er geen (unit)test-code is gevonden in de code-upload. Het is sterk aan te raden om in ieder geval voor de belangrijkste delen van de functionaliteit
automatische tests gedefinieerd te hebben om ervoor te zorgen dat eventuele aanpassingen niet voor ongewenst gedrag zorgen.

F.2. Second evaluation

In de tweede upload zien we dat de volume van het systeem is gestegen terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven. Het systeem scoort dus nog steeds 2 sterren, wat betekent dat het systeem ondergemiddeld onderhoudbaar is.

Als we deze upload met de eerste vergelijken, zien we nauwelijks verbeteringen op de eerder aangegeven verbeterpunten. Ten eerste is de naam van de “lib” component niet veranderd. Als deze jullie eigen geschreven code is, is het belangrijk dat de naam van de component anders dan “lib” is zodat er geen verwarring bestaat.

Wat betreft Duplication, Unit Size, en Unit Interfacing, scoort elk van deze eigenschappen nog steeds ondergemiddeld. Te veel overtredingen zijn te herkennen waar de principes van onderhoudbaarheid niet gevolgd zijn. Bovendien is het merkbaar dat dezelfde overtredingen die wij jullie met de vorige evaluatie hebben aangewezen blijven nog steeds te vinden.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie niet meegenomen zijn in het ontwikkeltraject.

F.3. Feedback interpretation and actions

The SIG evaluations have been negative and it seems that we have not taken notice of their critique, but our design decisions have been well considered and can we can justify them, therefore the second evaluation has not changed much from the first. The main reason that the SIG evaluations are negative is because we have build the system on M-industries platform and we have taken over the conventions that M-industries use. The system is not build in an object-oriented way, but in a model-driven way. This means that the IDE model has been the leading factor in the project. The code that we have been able to send to SIG excludes all the platform code and the code written in Alan, but is closely tied to the platform and the Alan models. We have used the ”lib” folder name because this is a convention used in the platform of M-industries. The majority of the code send to SIG is tied to the IDE model. Another convention in the MDD approach of M-industries is that if you write code against a model, the rule is to have a separate function for each component in the model. If the model contains a large component, then you would need a large function. In the IDE model we have very large component, namely the editor state. A function that walks the editor state can therefore be expected to be very large. This is probably why our code scores low on Unit Size. The example given by SIG of the method in the fillProject.js file is an example of this. It makes no sense to split the function, because outside of the context of the ide argument of the function, the code in the function has no meaning.

The example given for Code Duplication in the editorApi.js file can also be justified in this way. Yes, there is duplicate code, but the context of the code is different. Writing a separate function for those callbacks would actually decrease maintainability, because in a later stadium it might be necessary to differentiate the code for the different contexts and then the function has to be deprecated. The code follows the IDE model and if there are duplicates
in the model, then a separate component should be created in the model. Only then it would be justifiable to write a separate function. In this case it was not possible to write a separate component, so there might be some duplicate code when two cases are handled in the same way.

Concerning Unit Interfacing, this is mainly a concern in the file parse.js. The parser for the editor state is a very complex piece of code that has to deal with three layers of abstraction: the language model, a language instance, and the editor state, which models an instance of the language instance. This can indeed be very confusing, but that is not because of a lack of abstraction. On the contrary, this is because of a high level of abstraction.

Finally, there were no unit tests, and the reason for this has been well documented in chapter 4 about testing.


G. Infosheet

Title of the project
Browser-based editor for custom languages

Name and affiliation (Department and Group) of the TU Coach
Jan Hidders, assistant professor in the group of Web Information Systems

Name of the client organization
M-industries

Name and affiliation of the Client
Corno Schraverus, CEO at M-industries

Date of the final presentation
12 - 08 - 2015

Description
The initial goal of the project was to build an editor or IDE with support for M-industries model driven language: Alan. After a while the goal was to prove that functionalities these editors/IDEs have could be provided but built a model driven way. This shift was after a period of researching and finding that there were already numerous web based editors that could provide basic language support (i.e. syntax highlighting and such) if given some sort of a language definition. We found that building an MDD solution would prove to be more useful (since the language tools also work in a model driven manner) and more interesting. This choice also had impact on the development process, since MDD has very brief development cycles and so we had to set smaller targets. The biggest obstacle was to improve the performance in such a way that working with the IDE would be feasible, and this speedup was mainly gained in rendering and interface. The result was an IDE built with Alan to support Alan, which also meant testing it was easy: loading a file and seeing if the lexing, parsing and reference resolving steps succeeded. Though the product was not completely finished, it was a very strong proof of the versatility and power of MDD and specifically M-industries take on MDD: Alan. To use the IDE the language has to be made whitespace significant, so partial parsing can be implemented and full editability will not impact the experience that much (since every keypress will only trigger partial parsing instead of parsing the whole document.)

Members of the project team

**Elgar de Groot**
The adaption of the parser and reference resolver to work with our editor state were done by Elgar as well as a large part of the editor state model design including integrating the Alan language into the IDE.

**Stijn van Schooten**
The interface and interaction methods were built by Stijn and the server side file handler too. The improved render engine and markup languages were also created by Stijn.

All team members contributed to preparing the report and the final project presentation.
Contact persons
Elgar de Groot: elgar.groot@gmail.com
Stijn van Schooten: svsoke@hotmail.com
Jan Hidders: a.j.h.hidders@tudelft.nl

The final report for this project can be found at: http://repository.tudelft.nl
function lex(rawtext)
    lines := rawtext split on newline;
    current_line := root := new Line;
    forall line in lines
        if line is empty: skip line;
        words := line split on space;
        inString := false;
        parent := find parent based on indentation;
        current_line := new Line
        forall word in words
            if word contains un-escaped quotation marks
                if currently in a string
                    line.words append new Word(text + space + word, string);
                    inString := false;
                else
                    text := word;
                    inString = true;
                endif
            else
                if currently in a string
                    text := text + space + word;
                else
                    line.words append new Word(word, is keyword?);
                endif
            endif
        endfor
        parent.lines append current_line;
    endfor
Figure H.1: The simple lexer algorithm in pseudocode.
Figure H.2: The result of the lexer put into the try-out render engine. This instance has both a warning and a folded section.
Figure H.3: A mockup depicting the tree-based model.
Figure H.4: A mockup depicting the flat model.
Figure H.5: Sequence diagram depicting normal interaction with M-industries platform.

Figure H.6: Sequence diagram depicting modified interaction with M-industries platform for use in the editor.
Figure H.7: Sequence diagram depicting the old structure used to create a parsed editor instance from a document.
Figure H.8: Sequence diagram depicting a newer structure used to create a parsed editor instance from a document using the newer editor structure.
Figure H.9: Sequence diagram depicting the latest structure used to create a parsed editor instance from a document by mutating an editor instance.
Figure H.10: A simplified tree showing the decisions made every modification. The asterisk (*) shows where multiple options were simplified to one.
Figure H.11: Graph showing the speedup in performance with platform improvements for a large file.

Figure H.12: Graph showing the speedup in performance with platform improvements for a small file.
Figure H.13: A screenshot of how the IDE looks in its current state.