Useful feedback in the Ampersand parser

Gebruiksvriendelijke feedback in de Ampersand parser

Daniel S. C. Schiavini, Utrecht, The Netherlands
Maarten Baertsoen, Deinze, Belgium

Student numbers 851102873 and 850044695

Supervisor: Dr. Bastiaan Heeren
Examiner: Prof.dr. Marko C.J.D. van Eekelen
Customer: Prof.dr. Stef Joosten

Open Universiteit Nederland
Faculteit Management, Science and Technology
T61327 – Afstudeerproject bachelor informatica

June 25, 2015

Abstract

Ampersand is an approach for giving business rules a larger role in the software development process. The Ampersand compiler allows users to write business rules in a domain specific language (ADL) and process them into design artifacts, documentation and software prototypes. As the project becomes larger, users have significant issues with error messages of bad quality generated by the tool. Mainly for new users, errors are a large obstacle and cause frustrations.

Our assignment, given by Prof.dr. Stef Joosten, is to design and implement user friendly feedback for Ampersand. In order to improve the errors, Daniel Schiavini researched the qualities of good errors and the options for parsing within Haskell. The conclusion was to rewrite the parser using another parsing library (Parsec). In parallel, Maarten Baertsoen researched business rules and the Ampersand approach more thoroughly.

To guarantee the quality of the software delivered, we implemented a new library for testing the parser automatically. We added the possibility to 'pretty print' the parsing tree back into ADL code. Several efforts have been made to improve the code quality in the aspects of readability, extensibility, maintainability, documentation and performance.

Besides improvements in the parser, we have studied the ADL grammar itself. The grammar documentation was not up-to-date, so we reverse engineered the code and determined the recognized language. The updated grammar is now available as code annotations and documentation. In order to make the grammar more clear, performing and unambiguous, refactorings were applied without changing the language accepted by the parser.

The new parser is now merged with the main repository and is officially in production. Finally, an analysis on the quality of the errors showed that the old parser gave good errors in 22% of the time, while the new parser improved this percentage up to 82%. Bad quality messages went down from 21% to 1%.

In this thesis, we will show how the new parser achieved its goals and how this will allow for Ampersand to continue growing. This will save time and effort for students, researchers and commercial users.

Samenvatting

Ampersand is een methode om bedrijfsregels een grotere rol te geven tijdens softwareontwikkeling. Bedrijfsregels worden geschreven in een scripttaal (ADL) en door Ampersand gecompileerd naar ontwerpartefacten, documentatie en prototypen. Naarmate het project zich ontwikkelt, krijgen gebruikers steeds meer last van de slechte kwaliteit van de gegenereerde foutberichten. Vooral nieuwe gebruikers worden hierdoor gehinderd en gefrustreerd.

Onze opdracht, gegeven door prof.dr. Stef Joosten, is om betere gebruikersfeedback voor de parser te ontwerpen en te implementeren. Eerst hebben wij onderzocht wat goede feedback inhoudt en hoe dit in Haskell geïmplementeerd kan worden, met als conclusie om de parser te herschrijven met Parsec. Daarnaast hebben wij een onderzoek uitgevoerd naar het gebruik van bedrijfsregels en de Ampersand-aanpak.

De documentatie van de parser en de ADL-grammatica was niet up-to-date, waardoor wij reverse engineering hebben toegepast om de documentatie vast te leggen. Door refactoring hebben wij de leesbaarheid, uitbreidbaarheid, onderhoudbaarheid, documentatie en performance van de parser en de grammatica verbeterd, zonder de ADL-taal te beïnvloeden. Een automatisch testsysteem is geïmplementeerd met de mogelijkheid om parsebomen te prettyprinten als ADL-code.

De nieuwe Ampersand parser is inmiddels geïntegreerd en in productie genomen. Onze analyse toont aan dat de goede foutberichten van 22% naar 82% zijn gestegen, terwijl slechte fouten zijn gedaald van 21% naar 1%.

In deze scriptie laten wij zien hoe de nieuwe parser de doelstellingen behaalt en hoe deze verbeteringen het mogelijk maken dat Ampersand kan blijven groeien. Deze verbeteringen zullen tijd en inspanning besparen voor studenten, onderzoekers en commerciële gebruikers.

Contents

List of Figures

List of Tables

1 Introduction

Business rules promise to improve the quality and efficiency of the requirements elicitation process in information system projects. Once the business rules are consistently and coherently defined, they form a knowledge repository used for the provisioning of information within organizations. However, there used to be a large difference between the rules as seen by the Business Rules Manifesto and the way these were supported by software systems [4]. In practice, the methodologies supporting business rules in the domain of requirement engineering often focused on theoretical approaches, required expert knowledge and lacked the supporting tools to be used in commercial scale. With this issue in mind, the Ampersand approach was developed to tackle these imperfections, giving focus to rules formulated in agreements instead of actions [4, 5].

Ampersand is an approach for the use of business rules to define business processes. Users describe the business rules in a formal language (ADL) and Ampersand compiles those rules into functional specification, documentation and working software prototypes. As the project becomes larger, users have significant issues with error messages of bad quality generated by the tool, especially by the parser. Mainly for new users, errors are a large obstacle and cause frustrations.

Our assignment, given by Prof.dr. Stef Joosten, is thus to redesign and implement a new parser for Ampersand, providing user friendly error feedback [1]. In order to improve the error messages, Daniel Schiavini researched the qualities of error messages and the options for parsing within Haskell. By understanding what kind of messages the Ampersand users expect, we have developed a new parser for Ampersand based on the Parsec library. The main conclusion was to rewrite the parser using another parsing library, namely Parsec. In parallel, Maarten Baertsoen researched business rules and the Ampersand approach more thoroughly.

In this thesis, we will show how the new parser is strongly improved and how this will allow for Ampersand to continue growing. This will save time and effort for students, researchers and commercial users.

Several important aspects of this project are presented. To guarantee the quality of the software delivered, we implemented a new library for testing the parser automatically. We added the possibility to 'pretty print' the parsing tree back into ADL code. Several efforts have been made to improve the code quality in the aspects of readability, extensibility, maintainability, documentation and performance.

Besides improvements in the parser, we have studied the grammar of ADL. The grammar documentation was not up-to-date, so we reverse engineered the code to determine the recognized language.

Also, refactoring was applied without changing the language accepted by the parser, in order to improve the parser's performance and its maintainability. The new parser improves both the error message quality and the software properties related to readability, extensibility and maintainability.

Finally, a thorough analysis on the error message quality showed that the old parser gives good error messages in 22% of the time, while the new parser improves this percentage to 82%. Bad quality messages went down from 21% to 1%.

1.1 Identification

This document is the Bachelor thesis of the project 'Useful feedback in the Ampersand parser'. The document is the final product of the graduation project for Daniel S.C. Schiavini and

Maarten Baertsoen, as specified in the project planning [2].

This document is part of the graduation project of the Computer Science Bachelor ('Bachelor Informatica') at the Open University of the Netherlands. The project 'Useful feedback in the Ampersand parser' has been executed in collaboration with our supervisor Dr. Bastiaan Heeren and the examiner Prof.dr. Marko C.J.D. van Eekelen. The assignment was given by Prof.dr. Stef Joosten, who researches how to further automate the design of business processes and information systems by the development of the Ampersand project.

1.2 Related work

In the project phase 3a (domains & techniques) and phase 3b (research context) [3], multiple references are made to related publications. The documents providing additional insights or achievements related to the topics we touched during this project are listed below for further reference:

- **Deriving Functional Specifications from Business Requirements with Ampersand**: This paper [38] written by Stef Joosten provides an overview of the goals of the Ampersand Approach and the way these goals are achieved. The Ampersand rules together with the examples helped us as a starting point to get to know Ampersand and to reflect on in case of doubts.
- **Parsec, a fast combinator parser**: A useful document [6] written by Daan Leijen about Parsec. The document gives a step by step guideline on how to use and implement a Haskell combinator library.
- **Bedrijfsregels in verschillende vormen**: The master thesis [7] from Pim Bos in which a comparative research is outlined between the Semantic Web Rule Language and the Ampersand Language, including their supporting tools.

1.3 Document overview

In this section we have given a short introduction to our graduation thesis. Now, in Section 2 we will depict the project's objectives and requirements. We perform a domain analysis in Section 3 and Section 4 (phase 3a) and in Section 5 we describe our investigation on the research context (phase 3b).

Phase 3c of the project started with an as-is analysis, described in Section 6. After the analysis, the design and implementation follow in Section 7, and the tests in Section 8.

Finally we give our next step recommendations in Section 9 and a conclusion in Section 10. Further information, including the bibliography and a glossary, can be found in the appendices.

2 Objectives

In this section we give an overview of the most important objectives of this graduation project, along with an introduction to the project and its context. The complete list of objectives as given in the beginning of the project is given in the project planning [2].

2.1 Ampersand project

In November 2003, the Business Rules Manifesto [39] was published by the Business Rules Group, with the main purpose of declaring independence for business rules in the world of requirements. The manifesto supports the vision of business rules as equivalent to requirements. This is considered a radical change on how people see the world of business architecture [8].

In December 2010, Stef Joosten, Lex Wedemeijer and Gerard Michels published the paper 'Rule Based Design', presenting the Ampersand approach. The approach puts the rules in the center, using these rules to define the business processes. Ampersand is named after the & symbol with the desire of realizing results for both business and IT, in an efficient and effective way.

In 2011, the Ampersand compiler was created as an open source project. Since then, the compiler has been improved and applied in both business and academic contexts. The Ampersand end-users write business rules in a domain specific language (ADL), and compile that specification into functional specification, documentation and working software prototypes. These rules are based on agreements between the different stakeholders.

The theory behind Ampersand has been thoroughly studied, and is based on mathematical concepts, e.g. relational algebra and Tarski's axioms. Using the Ampersand compiler, users write the requirements in ADL and generate all the system specification independent of the platform. The main advantage is that the requirements' consistency and traceability are always correct (and even provable), from the lowest level up to the front-end. The requirements are presented to stakeholders in natural language, guaranteeing that any business expert who knows the context can validate the requirements. Figure 1 depicts the artifacts generated by the Ampersand compiler.

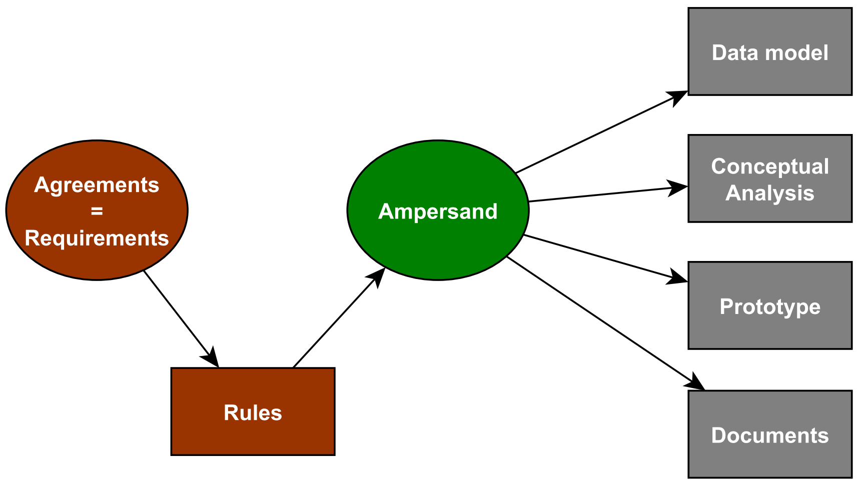

Figure 1: Ampersand generates several artifacts based on the business rules (see Section 3)

The Ampersand project is used in several environments, by different user groups. In a research context, the Ampersand project is part of the research on the use of business rules for software design. In an educational context, it is also used as the main tool in the course 'Rule Based Design' from the Open University of the Netherlands. Finally, the compiler is used in business environments to design and develop real world business software.

2.2 High-level architecture

The compiler developed for the Ampersand research project follows the traditional compiler architecture. It runs in several steps, hence the Ampersand compiler is also divided into several sub-components:

- **Parser**: This component receives the ADL code as input, and parses that code into a parse-tree (also known as P-structure).
- **Type checker**: The Ampersand type checker receives a P-structure as input and converts it into a relational algebra format, suitable for manipulation (also known as A-structure or ADL-structure). The semantics of ADL are expressed in terms of the A-structure.
- **Calc**: The Calc component receives an A-structure as input, and manipulates it according to the research rules, generating the functional structure (also known as F-structure). The F-structure contains all design artifacts needed to write a specification and generate the output.
- **Output components**: All design artifacts present in the F-structure are ready to be rendered. Several components use this data structure to generate the wished output. The output components currently implemented (and their output formats) are the following:
  - Atlas (HTML interface);
  - Revert (Haskell source);
  - Query (prototype generation);
  - Documentation Generator (Pandoc structure).

2.3 User-friendly parser

The end-to-end process of the Ampersand project, from compiling towards the generated artifacts, is according to the project's requirements. However, there is a major improvement topic identified in the first step, the parsing of the input scripts.

One of the main complaints from users is the quality of the errors generated by the Ampersand parser, making it hard for the end users to correct faulty ADL statements. Since the beginning of the project, the parser subcomponent never received special attention, and it has not been analyzed for improvements.

In order to generate better, useful and to-the-point error messages, it is assumed that a complete refactoring of the parser will be necessary [1]. The main challenge is to choose the correct kind of architecture and libraries. Note that discovering which errors are the most common and what user-friendly messages consist of, is an important part of the assignment. No list of good and undesirable error messages was available. Therefore it was up to us, as a part of the project, to judge which error messages were good enough and which were undesirable.

2.4 Other objectives

Besides the generation of better user feedback, the following objectives were also important for the parser:

- **Integration**: The new parser must interface with the remaining Ampersand modules. It must thus be implemented in Haskell.
- **Libraries**: Since different implementation options are available, it was important to choose the most suitable Haskell parsing framework.
- **Maintainability**: Well-written and maintainable code is a must for the Ampersand project, since it is an open-source project. The maintainability must be either preserved or improved; otherwise the parser will not be taken into production.
- **Tests**: Testing the parser well was a task for this project. The suggestion is to use testing tools to improve the tests, e.g. QuickCheck [1].
- **Pretty-printing**: This is important in order to test the parser.
- **Tools**: Within the Haskell community, several tools are popular to verify code quality and generate documentation. At least HPC, Haddock and HLint shall be used in the new parser.
- **Backwards compatible**: The new parser must process the same inputs and produce the same outputs as the old parser. Any changes to the parser output may have consequences to the rest of the system, and we have to adapt the rest of the system if we change the parser output.
- **Git/GitHub**: The changed software had to be integrated into the GitHub Ampersand project. The development itself happened in a separate branch of a separate fork, so that deliveries could be merged in a smooth way. This was an especially hard requirement for us, since we had no experience with Git.
- **Cabal**: The building system for Haskell had to be maintained as the building platform.
- **EBNF**: The syntax of the Ampersand grammar was specified in EBNF notation but was not up-to-date. Any changes to the syntax had to be documented in this notation. Updating the grammar in a separate file was one option. Another option was to add the EBNF as comment in the source code, in order to make clear that the grammar is implemented correctly. The last option is chosen to keep the EBNF close to the actual code. This stimulates the simultaneous maintenance of both the code and the EBNF notations.

On top of the project goals, we also wanted to help the university and other students with our results.

3 The Ampersand approach

The research on the Ampersand Approach is done by Maarten Baertsoen. The objective was to understand the vision of Ampersand in relation towards other formal methods within the same domain. It focuses on the strengths and weaknesses of formal methods for defining business requirements. The Ampersand approach is positioned within these formal methods based on the strengths and weaknesses of the different approaches.

3.1 The Ampersand methodology

3.1.1 Software requirements, the problem statement

In 1976, Bell investigated the domain of requirement engineering [9], sponsored by the Ballistic Missile Advanced Technology Center, with the goal to determine the magnitude and characteristics of requirement-related problems in software engineering and to identify techniques that could correct these issues. One of the main conclusions was that requirement errors were the most numerous and that these kind of errors are very time-consuming and, hence, costly to correct. In his conclusion, Dr. Thomas E. Bel advises to use methods and techniques during the requirement engineering process to ensure consistency within and between the requirements, such as the unique naming of objects and correct relations between the requirements themselves. In addition, he stressed that the applied methods must aid the verification and validation of the requirements.

Although the research is outdated from an IT point of view, the conclusion is actually still very relevant and therefore, methods and techniques to further improve the quality and consistency of the captured requirements are still a hot topic within the domain of requirement engineering.

An important classification of requirements is made by Joosten [38] and Borgida [10], into early phase requirements (the business requirements), and the late phase requirements (the functional specifications).

Formal functional specifications The initial focus to tackle the requirements issue, in which the research still continuous after 30 years, was on the functional specifications by the introduction of formal methods for functional specifications [40], which were based on a mathematical representation of the specifications resulting in the ability to analyze, validate and transform requirements into useful artifacts during the subsequent design and implementation phases.

As described by Luqi & Goguen [11], these formal methods for functional specifications had a positive impact on the reliability of the software development process, specifically for the purpose of specification analysis, transformation and verification [12]. This has resulted in the elaboration of several mature formal methods. The formal methods are categorized into three domains:

Algebraic languages describing the system in terms of types of data, mathematical operations on those data and their relationships, such as Z [13], VDM [41] and Alloy [14].

Model based languages using mathematical sets and sequences to express the system specification as a system state model such as OBJ [42], Larch [43] and LOTOS [15].

Hybrid languages combining features of both algebraic as model-based languages such as RAISE [16] and CafeOBJ, an enhancement of the OBJ language [17].

Business requirements Although formal specification methods had a positive impact on the software development process, it was recognized that these techniques were not sufficient. Concisely summarized by Yu and Mylopoulos [18], the formal specification methods focus on the 'whats' and the 'hows' of the desired system without an understanding of the 'whys' behind the system.

Within the social environment in which a software system will be introduced, several stakeholders have different goals, business requirements, and based on these, they will express, often imprecise and inconsistent, expectations. Only by the correct understanding of their business requirements, to a feasible extend, the requirement engineers will truly feel the 'whys' needed to derive the correct 'whats' and 'hows' to support these different goals.

Several early phase requirement modeling languages, RMLs, were introduced and typically included [10]:

An ontology of requirements describing the needed information to capture and the desired properties and behavior of of the to-be system, the view on the world from the perspective of the to-be system. This includes instances such as 'Entity', 'Activity' and 'Assertion' [19].

Modeling primitives, to model the concepts and the relations within the ontology.

Methods, sometimes automated, to verify consistency and to perform additional analysis to verify if the stated requirements will satisfy the business expectations.

Early RMLs such as RML [19], provided initial methods but suffered from drawbacks such as extensibility as the provided ontologies were rather fixed meaning that no new notions on par with the existing instances could be addressed. CML [44], which evolved to Telos [20], contained already additional flexibility.

KAOS [21] introduced the notion of 'stakeholder goal' where i* [22] even differentiated between independent and interrelated, joint goals. Further research to introduce the concept of goal priorities is ongoing for which a new abstract requirements modeling language 'Techne' [10] is designed, based on the CORE ontology for requirements [23]. Techne will provide the framework for new RMLs containing methods to compare candidate solutions and their compliance towards the business requirements, a feature that will come in handy as many software engineering projects nowadays include COTS package based solutions.

Limitations and frequent issues Besides the significant improvements and their positive impact in software engineering projects [38], several striking drawbacks related to the use of formal methods are identified [11]:

Communication Communication between the business users and the requirement engineers based on the mathematical model is difficult as the business users are not familiar with mathematical notations. Although the functional specifications are analyzed to make sure they are consistent, it still offers no guarantee that the business requirements are consistent and transformed correctly into functional specifications as they cannot be correctly understood by the end users.

Typical experience and knowledge of developers Software developers are not used to write code based on mathematical models, or even lack the skills of higher mathematics. It requires extensive training for these developers to understand the mathematical models and to develop efficiently in these kind of software projects.

Theoretical approaches Many formal methods are theoretical and their appliance is demonstrated by means of very simplified example, but when they are effectively used in practice, the gap between theoretical specification and practical coding appears to be problematic, not to say impossible.

Agile development Building a formal specification of a complete system is sometimes perceived as not flexible towards agile development techniques in which a system is engineered incrementally.

Package based development Many large software projects nowadays are using package based, COTS, solutions that are configured to fit the needs of the users. In such projects this package is mostly kept as standard as possible, meaning that no or little development is added and the functionalities of the package are used as implemented. In most commercial packages, no specific formal information is provided that allow the use of formal verification and validation techniques. The same goes for cloud solutions which are offered as-is without any, or only very limited, room for changes.

From business requirements towards functional requirements Early and late phase requirement methods tend to focus on their specific domain, business of functional requirements. Not many methods provide a means to verify the translation from business to functional requirements.

Supporting tools Most formal methods do not have supporting tools making it very cumbersome to use them in larger scaled projects, even when there are supporting tools available, they often lack a suitable user interfac, which threatens the practical use of the methods.

3.1.2 The goals of the Ampersand methodology

The Ampersand Methodology was crafted in 2007 by the inventor Joosten [38], with the vision to provide an answer to several of the main drawbacks of using formal techniques in software engineering (listed in Section 3.1.1). The Ampersand Methodology is developed to provide a method to unify the informal process of capturing the needs of users with the formal process of specifying an information system and to provide a formal translation method between both processes, including the necessary tools to ensure that the methodology is useful in real life projects. Special attention is given to the transformation and verification of the business requirements into functional specifications.

Summarizing, the methodology presents the means to structure and present requirements in such a way that they can be validated by end-users and be interpreted unambiguously by system engineers after an automated transformation into functional specifications and design artifacts while the consistency between both is guaranteed.

The Ampersand Methodology addresses several goals to achieve this vision [38]:

Communication To assure that the stakeholders can correctly understand and validate their needs, the requirements must be documented in a natural language to enable them, without any requirement engineering knowledge, to validate the formal system requirements. On the other hand, the requirements must be structured into formal functional specifications to make them useful during the actual software development phase and to benefit from the advantages of formal methods.

Completeness A software system in which not all requirements are supported is useless. The Ampersand language must be fully declarative, meaning that all the requirements must be supported in the method to assure that all business requirements, relevant to the subsequent software system, are accommodated correctly in the system specification.

Consistency One of the main goals of the Ampersand Methodology is to guarantee consistency: each specified requirement must be applicable to, and respected by, all processes in the context of this requirement. The methodology needs to provide the means to assure this consistency.

Traceability The requirements must be traceable in such a way that end-users can trace functions back to the stated business requirements, allowing them to fully understand the reason why these functions are defined.

Supporting tools In order to facilitate the adoption of the Ampersand Methodology outside an academic environment, Prof.dr. Stef Joosten defined the additional goal that the Ampersand Methodology must be accompanied with supporting tools to support the requirement engineers and this by creating design artifacts to be used by the software developers.

3.1.3 The Ampersand approach

In theory, the Ampersand Methodology can support any set of business rules, a language expressed in relational algebra, that satisfies the Ampersand Methodology axioms [38]. An instance of such a language, a specific grammar, is specified to support the practical use of the Ampersand Methodology: the‘Ampersand Definition Language’, ADL. This language is accompanied with a supporting ADL tool to generate design artifacts, including a prototype, based on a set of business rules. The ADL tool, in which input according to the ADL language is processed, is a specific approach to guarantee the achievement of the Ampersand goals.

Communication The innovative aspect of the Ampersand methodology is that the business requirements in the ADL language are represented as ‘business rules’ in natural language using relational algebra. A business rule must be seen as a business requirement in the form of an invariant to be satisfied by the business. By using natural language, the business rules can be easily understood, and validated, by the business users.

The business rules are transformed in an automated way, by the ADL tool, assuring the correctness of the functional specification based on the relational algebra of the business rules and the formal nature of the ADL language.

Once the functional specifications are generated, the ADL tool will transform them back into business rules for business user validation. When the re-translated requirements are then still correct for the end users, the system engineers have the assurance that the functional specifications are fully consistent with the business requirements.

Systems engineers and business users can better communicate as they both have a specific view on the same requirements formatted in a way they fully understand.

Completeness Early methodologies to define business requirements lacked the power to add new notions besides the pre-defined instances. The ADL language is designed as a purely declarative language to avoid this drawback by featuring user specified rules, design patterns and signaling constructs.

In the ADL language, business rules are specified without specific pre-defined actions to avoid that they become too narrow, compromising the declarative aspect of the ADL language.

Consistency Expressing each business requirement as a business rule using relational algebra provides a mathematical foundation to check all rules against each other and to identify inconsistency between two or more business rules. Checking each rule on its consistency with the full set of defined business rules manually is quite demanding and would compromise the efficiency of the method on real life projects. The consistency check is therefore automated by the ADL tool.

Traceability Business requirements have a one-to-one relationship with business rules offering traceability back from a specific rule, specification, to the business requirement making it possible for the end-user to correctly identify the reasons why the business rule was identified.

Supporting tools The practical use of the Ampersand Methodology is supported by the ADL tool built in the functional programming language Haskell. This tool produces a wide range of functionalities and design artifacts to support the validation, consistency checks and the subsequent software development steps:

Business rules and consistency issues The declared business rules in the ADL Language are compiled and typed checked. This is realized by using Swierstra's combinator package for the compiler [24] and the AG-preprocessor by Dijkstra and Swiestra for the type-checker [25].

Data model Class diagrams and entity-relationships are created using the the GraphViz package [45].

Service catalogue The possible services such as create, get, update and delete, are specified formally for all classes and relationships that are specified in the business rules. This formal representation together with the pre- and postconditions of the service make it possible that several developers can program the services independently of each other. As the tool generates the full set of services for every class or relationship, it is up to the requirement engineers to determine the set of services that actually need to be implemented.

Function point analysis A function point analysis providing an insight on the complexity of the system to be built is generated according to the IFPUG guidelines (`http://www.ifpug.org/`). This degree of complexity can be used to estimate the remaining system development effort taking into account the impact of the Ampersand tool as an accelerator.

Software prototype A remarkable aspect of the Ampersand tool is that it makes it possible to generate a working prototype. The prototype is generated as a web application and uses a MySQL database.

Training Although the Ampersand Methodology, including ADL, is not yet widely adopted in the domain of requirement engineering, the methodology, including the supporting tools, is educated by the Open University of the Netherlands, in the course 'Ontwerpen met bedrijfsregels'. A specific site is dedicated to the methodology, as well for self-tuition or co-development purposes, (`http://wiki.tarski.nl/`).

3.2 Ampersand in practice

3.2.1 Getting started

To use the Ampersand tools, several components need to be installed:

Ampersand installer is an installation package containing the full Ampersand tool. This package can be downloaded from the Ampersand home page (`http://wiki.tarski.nl/`). The necessary installation instructions are provided on that same website.

MiKTex, a LaTeX compiler used to generate documents which are presented in pdf format.

Graphviz is used to generate the pictures, such as the class diagrams and the entity-relationship diagrams.

XAMPP can be used in case your environment does not support an HTTP/PHP server and/or a MySQL database.

After some post-install actions, the Ampersand tool can be started in the MS-DOS command prompt in MS Windows. No graphical user interface is yet available. All commonly used command line parameters are listed on the Ampersand web page.

3.2.2 Requirement gathering

The business rules are recorded in a plain ASCII text file, created using a simple text editor such as Notepad++. No specific user interface supporting on the fly checks on structure and types, e.g. brackets and names, is yet available.

3.2.3 Practical usage

The correct installation and usage of the Ampersand tools and the additional tools requires some IT awareness, especially in case some unexpected errors occur when the Ampersand tool uses the additional tools.

To get used to the specific syntax of how business rules are defined in the ADL language requires some training. Besides the base syntax, a specific patterns hub is available, to support new and seasoned ADL users, which can be used for training, inspiration and re-usage.

As already mentioned in the introduction of this document, the feedback generated from the Ampersand tool, in case issues are detected in the input file, with respect to the syntax, appears to be insufficient and needs to be revised.

3.3 Conclusion

Based on the theoretical approach and the practical usage, this section provides some conclusion regarding the methodology as well as the supporting tool. All strengths, weaknesses, opportunities and threats mentioned in this document are summarized in the SWOT diagram in Table 1.

3.3.1 Future road map

Not surprisingly, several of the current weaknesses of the methodology will be addressed in the future Ampersand road map to strengthen the power of the Ampersand Approach:

Strengths	Weakness
- End-2-end approach and consistency assurance using relational algebra - Availability of supporting tools - Communication is aligned to the knowledge, skills and role of the receiver - Reduced effort → cost efficiency* - Thorough training available - Automated creation of prototypes for stakeholder alignment - Creation of design artifacts as an accelerator - Usability on real life projects	- No user interface - No distinction between mandatory and optional requirements - No specific support to support Agile approaches - Feedback of errors during the compilation process are insufficient - Maintainability of the code can be improved
Opportunity	**Threats**
- Evolution to bigger, more complex and more integrated projects increases the need to assure that all requirements are consistently respected whilst the effort needed to assure this consistency raises exponentially on these kind of projects. No, or very little, methods can offer support to deal with this. - Emerging techniques such as cloud based solutions and COTS products requires another requirement engineering techniques than the formal methods	- Stakeholders in software engineering projects want a greater control over the articulated requirements - Short time to market projects using agile approaches and incremental deliveries - Cost reduction is stressed in almost each software project

Table 1: SWOT Analysis
* cost reduction measured on real life projects in which the Ampersand method as used [38]

Improved generator Instead of working prototypes, a more powerful generator will be built that can generate complex and full-scale systems.

Structure repository The requirements engineer will be offered a wide range of standard structures as a starting point to further tune to the specific project requirements. This will allow the requirements engineer to focus more on the specific requirements needs.

Graphical user interface A GUI will provide a full blown working environment for the requirement engineer providing real time syntax and other structure issues. This will reduce the amount of errors found during compilation and hence increase the efficiency and user satisfaction. A good compiler that provides useful feedback will remain important to tackle the issues not found in real-time during the requirement editing.

Agile approach Short functional iterations, in which prototypes are generated, will be supported to embed the base principles of agile approaches.

Feedback of compiler messages As part of this project, the compiler will be revised to provide clear and useful errors messages.

3.3.2 Concluding remarks

Without doubt, the application of formal methods can enhance the quality of the actual delivered system as the quality of the full software engineering process. Besides the positive effects, most of the formal techniques include some disadvantages due to which these methods are often neglected. During the elaboration of the Ampersand method, Joosten clearly focused on a method, and supporting tools, that can be used on large real life projects. The use of relational algebra to assure end-2-end consistency, the formatting of requirements in natural language for business user alignment together with the supporting tools which automate, otherwise time consuming, checks and generate useful design artifacts including working prototypes is ground-breaking.

The Ampersand methodology has a large potential to gain an important position next to other formal methods. Two identified weaknesses, the quality of the feedback and the maintainability of the code will be tackled in the next phases of the graduation project of Daniel S.C. Schiavini and Maarten Baertsoen.

4 Parsing libraries & friendly errors

The research on parsing libraries and user-friendly error messages was done by Daniel Schiavini. The objective was to gather enough technical information to support the design and implementation of the new Ampersand parser. It focuses on knowledge acquisition in two interrelated fronts: a search for the parsing library best suited for this project and defining what constitutes good error messages.

4.1 Generators vs. combinators

Parsing is sometimes divided into two stages: lexical analysis (separating the source text into tokens) and parsing itself (constructing a parse tree from these tokens). Tools such as the ones analyzed here can perform both lexical analysis and parsing. However, sometimes the tools can be more efficient when supported by a separate lexer (e.g. Alex).

Grammars associated with a formal language are described as a set of production rules. Since these rules are formally defined, a series of mathematical constructions can be used to manipulate and describe the grammar. The Ampersand Definition Language (ADL) is specified in a grammar in the Extended Backus-Naur Form (EBNF).

Generally, there are two options for constructing a parser: The first option is to construct the parser in the language of choice, i.e. Haskell for this project. Another possibility is to use a domain-specific-language (DSL) to describe the grammar and let separate software generate the actual parsing code. The two approaches and their advantages and disadvantages are described in the following subsections.

4.1.1 Parsing libraries

When programmers go down the path of building a parser directly in Haskell, building up a set of functions that support parsing is a natural consequence. Although it is possible to build these functions for each and every project [26], using a premade library is usually more advantageous, because of e.g. reduced effort, increased functionality, optimized performance and better documentation. The extra effort to learn the library is paid off by these advantages. In Haskell this is mostly done by providing monadic and/or applicative combinators to hold up extra information about the parsing state, and results in very elegant solutions [26]. Combinator libraries are often called Embedded Domain Specific Languages (EDSL), as opposed to an external, stand-alone DSL.

The parsers built in Haskell are mostly recursive descent parsers, wherein the parser runs top-down from a set of recursive calls. The program structure is then closely related to the production rules, supporting the readability of the program structure.

These parsers analyze the input text from Left to right and choose the Leftmost derivation in the grammar. Such parsers are called therefore LL parsers.

However, there are limitations related to recursive descent parsers. For instance, a common kind of production rule is a left recursive one, e.g. `term → term '+' term | digit`. In this case, the first thing the parser would do is call itself, resulting in an infinite loop. Fortunately, left recursive grammars can be converted into right recursive ones [27].

Therefore, LL parsers may require exponential time to run and are not able to guarantee termination. In order to guarantee termination and linear execution time, a recursive descent parser must be able to recognize which production rule to use by reading only limited amount of tokens. This is only possible for the class of unambiguous grammars without left recursion.

The EBNF for the ADL-language has no left recursion. This grammar can therefore be called an LL(k)-grammar, for which an LL(k)-parser can be created. The k between the parenthesis means that this parser needs a maximum of k tokens of look-ahead in order to choose a production rule.

4.1.2 Parser generators

As mentioned, another approach for building a parser is to specify the software's grammar in a specific notation and use a parser generator to create the actual parser source code. In the domain of context-free grammars, a widely used grammar notation is the Backus-Naur Form (BNF).

This research is focused in the Happy parser generator, which is part of the Haskell Platform since 2001. Happy inputs a file containing an annotated BNF specification of a grammar and produces a Haskell module with a parser for that grammar. Since it is possible to convert EBNF to BNF [46], specifying a Happy parser should not involve a lot of effort. It is important to know, however, that besides converting the EBNF to BNF, the annotation still requires effort and knowledge acquisition.

A parser generator is able to execute statical analysis on the input grammar. Besides, generators are often able to recognize more complicated grammars by running Left-to-right, and picking the Rightmost derivation, being called therefore an LR parser.

An LR parser with Look-Ahead is called an LALR parser. By performing the rightmost derivation of the production rules, an LALR parser is able to recognize more complex grammars. However, its workings are quite unintuitive, and understanding such parsers can be very hard [28]. That is exactly why LALR parsers are usually generated instead of built by hand.

Since understanding LALR parsers is hard, the syntax errors caused by incorrect input may be much harder to pinpoint and understand. The errors generated by LALR parsers are often not in high-level terms that the end users can understand.

Haskell compilers GHC and Hugs are both built with LALR generated parsers: Hugs is written in YAC and GHC is built with Happy. Later on, the Helium compiler was created for classroom-use because the mentioned Haskell compilers were not user friendly [29]. Other examples are the GCC compilers for C and C++, that started as LALR generated compilers and were remade to be recursive-descent parsers.

Although it is harder to understand its workings, the resulting source code for the generators is simpler and easier to maintain, as can be seen in Hulette's examples [47].

4.1.3 Conclusion

In the context of the new Ampersand parser, some advantages of building the parser in Haskell instead of using a generator, are:

- **Flexibility:** The programming language gives much more flexibility in coping with context-sensitive grammars.
- **Building:** The process is simpler since it is unnecessary to run a separate program to generate the parser.

- **Language:** Both the customer and the project members feel more comfortable working in Haskell than in an unknown DSL.
- **Errors:** The main objective of the project is giving useful feedback in the new Ampersand parser, and this seems much easier to achieve with a handwritten parser.

On the other hand, the advantages of using a parser generator, instead of handwriting the code, are:

- **Optimizations:** Because the parser is generated on-the-fly, the generator can apply optimizations that would otherwise be hard to implement.
- **Performance:** Bottom-up parsers are much more efficient because they are able to pack the code into state machines. This is even more valid when many parsing alternatives are available.
- **Static analysis:** The generator is able to do a lot more static analysis, while a library is only executed at run-time. E.g. programmers may only know of left recursions and non-terminations by testing the parser.
- **Documentation:** Since the DSL is basically annotated BNF, keeping the syntax diagrams up to date is much easier.

From the above advantages and disadvantages, it is clear that no universal truth exists in these matters. Although it is a difficult choice, the error messages are indeed the most important project target, so writing the parser by hand is the advised option. This choice also means that it keeps on being a task of the developers to update the documentation.

4.2 Haskell parsing libraries

In the previous section, the choice to use a combinator library has been taken. In this section, two libraries will be compared: the Utrecht University parser combinator library (UU-parsinglib) and Parsec. Other libraries (e.g. Attoparsec, Polyparse) are out of the scope of this research.

4.2.1 Utrecht University parsing library

The UU-parsinglib is a combinator library created by Doaitse Swierstra in the Utrecht University. This library is used in many mature projects and has more than four thousand downloads in the Hackage package manager. Two versions of the library are available:

- The UUlib library is older and more limited, dating back from 2005. It is used in most of the projects distributed by the Utrecht University, and in the current Ampersand parser.
- The new version, named UU-parsinglib, dates from 2008, has several improvements and much simpler internals because of GATD. This version is considered in the rest of the document because of its performance [48] and the author's recommendation [49].

The documentation is mainly in Haddock format and in a paper from 2009 [24]. The implementation is open source. The new version of the UU-parsinglib provides combinators that include error correction and a monadic interface. Errors are recognized and corrected automatically if the programmer wants so, but the error reporting is customizable. It also supports grammars that are not context-free and even ambiguous grammars (provided the user accepts exponential run-time).

Finally, the UU-parsinglib runs online, i.e. it returns parts of the parsing three as soon as they are ready. This gives programmers the ability to do lazy parsing.

4.2.2 Parsec

Parsec is a monadic parsing combinator library created by Daan Leijen, while also working at the Utrecht University. It seems to be the most popular combinator library in the Haskell community, with more than 200 thousand downloads in the Hackage package manager.

Parsec is designed to be simple, safe and well documented industrial parser library. Besides, there has also been some work done on the performance and error messages. The documentation of Parsec and its documentation tends to be better than that of the UU-parsinglib because of a larger user base.

4.2.3 Monadic vs. applicative interfaces

Monads allow sequences and state to be saved during the parsing, and had become the most common way of building Haskell parsers. Until, in a paper from Swierstra [30], a parsing library was published with an alternative interface. McBride and Paterson presented this alternative as a generalization of monads, calling it 'applicative functors' or 'idioms'. Applicative interfaces are less convenient than monads but are more widely applicable [31].

Applicative parsers do not depend on the run-time values – they are not dependent on the context. This means that it is then possible to analyze and optimize the parser before executing it. Swierstra goes so far to say that this works as a run-time parser generator [30]. On the other hand, the lack of context means that only context-free languages can be implemented this way [32].

Both the UU-parsinglib and Parsec have monadic and applicative interfaces. In the applicative interface, the UU-parsinglib also adds error correction: After detecting an error, the library will correct it by adding or deleting tokens. It then generates an appropriate error message and continues with the rest of the program. By using this interface, the parsing will thus always succeed.

4.2.4 Conclusion

The following differences have been found between the two considered libraries:

- **Documentation:** Parsec's documentation seems to be more extended and well maintained than that of the UU-parsinglib. Several Parsec tutorials can be found on the internet (e.g. [50]). Besides, the UU-parsinglib documentation is mostly generated from code annotations.

- **Support:** Since Parsec is much more used, online support can be more easily found. For example, the website Stack Overflow currently has 14 questions about the UU-parsinglib and 301 about Parsec.

- **Static checking:** None of the libraries is able to do static checking. However, the UU-parsinglib has more possibilities of grammar analysis in its applicative interface.

- **Precedences:** The Ampersand parser is currently built with the UU-parsinglib with great satisfaction. This both means that the library has enough features and that it is known by the other Ampersand developers.
- **Error reporting:** No literature has been found with a comparison of the errors generated by the libraries. However, many publications affirm that the generated errors from both libraries are great [29, 24, 30, 32].
- **Error recovery:** When a parsing error is found, a Parsec parser stops immediately. A parser built with UU-parsinglib, however, corrects the error and continues parsing. Error correction is good because the parser always succeeds. On the other hand, a big list of errors can also overwhelm the user [33]. Finally, to perform the corrections, the parser needs to make assumptions based e.g. on statistics; these assumptions cannot always be correct.
- **Fine-tuning:** According to the Helium development team, Parsec's possibilities for error fine-tuning are greater [29]. However, to apply optimizations it is necessary to know the internal workings of the parser [24].
- **Backtracking:** Parsec works with more traditional backtracking algorithms [32] that can often lead to high space consumption [24]. Backtracking must be manually activated, though, with it's *try* combinator. This combinator is considered harmful and can be easily misused [51]. The UU-parsinglib, on the other hand, uses breadth-first lazy parsing [24] so such combinator is not necessary.
- **Performance:** The Parsec library seems to have better performance [48], but the difference is small and is not expected to make a considerable difference in the small ADL scripts.
- **Maintainability:** No significant difference has been found in the maintainability of the two analyzed libraries. Note that the programmers working on Ampersand are already familiar with the UU-parsinglib. On the other hand, there is more support and documentation available for Parsec, so it can also be seen as more future-proof. The responsibility for the maintainability still lies on the hands of the programmers building the parser.
- **Origin:** Both libraries are originated at the Utrecht University. In 2003, a Haskell compiler focused on user friendliness, Helium, was published from the same university. Knowing both libraries very well, the authors made the choice to use Parsec because of the possibilities of error customization [29].

Considering these differences, a deeper analysis of error messages is given in the next section. The actual advice on the library choice is delayed until Subsection 4.4.

4.3 User-friendly error messages

4.3.1 Parsing errors

When a parser is executed and is unable to recognize the input text, an error is raised. The main problem when this happens, is that the parser cannot know for sure what the programmer meant to write. Only the programmer can know exactly what the purpose of the invalid input was.

However, parsers should be able to recognize the most common errors, and support the user to correct them. Spenke et al. [34] discuss the following assumptions regarding parsing errors:

1. An incorrect program is very similar to a correct one;
2. An error is very soon followed by a correct piece of program;
3. There are symbols (e.g. keywords) that are omitted quite rarely, while some less important symbols are more frequently omitted (e.g. semicolons);
4. There are some very reliable symbols which most likely do not occur by accident, but always in an specific context (e.g. then being part of an if statement);
5. If an error cannot be corrected by deletion and/or insertion of a few symbols, the reason is often a complete, misplaced syntactic unit, such as a whole expression where only a single constant is allowed;
6. A frequent reason for syntax errors is typing errors. In addition, similar basic symbols, such as round and square brackets may easily be confused.

4.3.2 User-friendliness

There is no formal definition of what user-friendly error messages are. Indeed, each kind of user is different and may expect something else from the system. For example, an expert programmer can understand much more of technical implementation details than an inexperienced student.

Therefore, it may be very important to be able to fine-tune the errors shown. The developers of the compiler may want details of the inner workings of the system (e.g. the parse tree). Students, on the other hand, may just want to see the most likely cause of their mistake.

4.3.3 Implications for the Ampersand parser

The following items have been identified as important in the error messages generated by parsers:

Location When an error is detected, it is crucial to point out where in the input text the error has been found. If the location is incorrect, the user will have to search through his/her entire program to find the syntax error [29, 24, 30, 32].

Production rules Listing which product rules are applicable at that moment can help the user to choose the correct one [29]. However, students have been reported to get overwhelmed by the huge list of terminals currently generated by Ampersand in the case of errors [33].

Misspellings Very often, errors happen because the users mistype keywords and/or symbols. Tokens that are invalid, but very close to valid input and often misspelled, should be recognized by the parser [29, 34].

Error recovery When an error occurs, it is appropriate that the parser is able to recover from that error. This way, the parser can continue analyzing the input [30, 34]. In some cases it may be wished to display all the different errors, and in some cases only the first one.

Error output Most compilers output errors as plain text. However, richer formats may be more appropriated for communicating with the user, e.g. syntax diagrams or HTML. Other formats are only possible if the development environment can correctly display the errors.

4.4 Conclusion

In Subsection 4.1, the advice was given to use a combinator library for the new parser of Ampersand. The main reason to avoid the parser generators is that it is hard to generate useful feedback. Then, in Subsection 4.2, two different libraries have been investigated. In Subsection 4.3, it was made clear that besides generating good messages, those messages should also be customizable.

Therefore, the advice of this research is to use the combinator library that offers the highest level of customization in error messages, Parsec. Although the UU-parsinglib seems to also be a very good choice, the experiences from the Helium compiler [29] should be also considered. Besides, the Parsec library seems to offer better support.

A list of important consideration points has also been collected through the literature and can be found in Subsection 4.3. Finally, other factors besides the errors are also very important for giving useful feedback. One of the most important factors is that the documentation should be always kept available, up-to-date, clear and concise.

5 Research contexts

In this project phase we identified the research contexts related to the Ampersand approach as a whole. After identifying the different contexts and understanding them, for each identified context we described the influence of this project on the bigger picture.

The starting point of the investigation is done through literature research, allowing us to understand the actual research context. During the literature study, key questions are formulated which are presented afterwards, together with the actual literature study, to a consult researcher. For this investigation, we consulted the expert researcher Dr. Lloyd Rutledge, who was willing to be part of an interview.

Our conclusion is that the new Ampersand parser has a very small influence on its research context for business rules. The parser does not offer any new features or improvements to the language. However, we expect that the new parser will improve user experience, allowing commercial, research and educational users to do their work faster and more efficiently.

The complete research report is available as a separate document [3].

6 The old parser

In this section, we analyze the old Ampersand parser in order to understand its workings and signal the improvement points.

6.1 System overview

The requirements of the new parser have been described in the project planning [2]. Basically, both the old and the new parser must be able to make a conversion from a text file (ADL-script) to a parse tree (the P-structure). Figure 2 depicts this data flow.

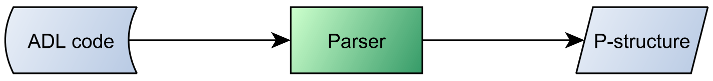

Figure 2: Relevant data flow for the Ampersand parsing component

Often, the parsing component is separated into a lexer/scanner (that converts text to tokens) and the actual parser (that converts the tokens into the parse tree). Since this separation is considered beneficial for both maintainability and performance [32], we assumed from the beginning that the new Ampersand parser would be separated in this way. This is depicted in Figure 3.

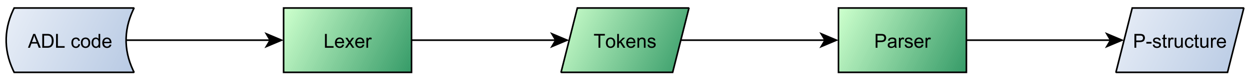

Figure 3: Data flow for the Ampersand lexing and parsing components

In order to take the next steps and understand how the parser can be designed, we first take a look at the current errors in Subsection 6.2 and we investigate the grammar in Subsection 6.4. Afterwards, we analyse the lexer in Subsection 6.3, the parser in Subsection 6.5 and the parse tree in Subsection 6.6.

6.2 Errors

In this section, we analyze the error messages given by the old parser. The results of this analysis are compared with the new parser in Subsection 8.2.

6.2.1 Error message qualification

The user friendliness and correctness of an error message is a subjective topic and therefore we need to start with a definition to objectively judge the quality of an error message. In order to identify the objective aspects of an Ampersand parser error message, we decided to use an existing definition. Yang et al. suggest a manifesto to measure the quality of reported type error messages [35, 36]. Since the definitions of this manifesto are not strictly dependent on type errors, we consider it also well suited to parsing errors. According to this manifesto, a good error report should have the following properties.

- **Correct:** An error message is emitted only for illegal programs, while correct programs are accepted without an error report.
- **Accurate:** The report should be comprehensive, and the reported error sites should all be relevant and contribute to the problem. Moreover, to understand the problem, only these sites should be inspected.
- **Intuitive:** It should be close to human reasoning, and not follow any mechanical inference techniques. In particular, internal type variables should be introduced with extreme care.
- **Succinct:** An error report should maximize the amount of helpful information, and minimize irrelevant details. This requires a delicate balance between being too detailed and being too terse.
- **Source-based:** Reported fragments should come from the actual source, and should not be introduced by the compiler. In particular, no desugared expressions should be reported.

To expand on the error manifesto, we consider the following characteristics to be part of intuitive parsing errors:

- **How does the provided error description outline the discovered error:** Providing the user with a good description of the encountered syntax issue will support a fast error resolution. When the issue is vaguely described without pinpointing the exact issue, the error resolution will be time consuming.
- **Pinpointing the correct error**: An error can lead to multiple subsequent issues. These issues are, however, irrelevant for the user and the Ampersand parser should provide the exact origin of the issues.
- **Quality of the hint:** The message can provide a hint for a solution together with the error message to support the user with the error resolution.

Based on these objective properties to judge the quality of the parsing errors, we define the following criteria to distinguish between good, bad and average (but acceptable) error messages:

Bad error message: A message is considered to be bad if one of the criteria below is fulfilled:

- **Incorrect:** An error message is given while the program is correct or no message is given for an incorrect program.
- **Inaccurate:** The parser gives a source position that is irrelevant to solve the problem. The given position has a deviation of more than one line or ten column positions from a relevant position.
- **Unintuitive:** The provided error description is useless for the user to determine the actual error, or it appoints an error without any relation to this main error.
- **Not succinct:** More than three distinct errors are mentioned by the Ampersand parser.
- **Not source-based:** The given error does not come from the actual source.

Acceptable error message: A message is considered to be of average (but acceptable) quality if one of the criteria below is fulfilled:

- **Inaccurate:** The position has a deviation between five and ten column positions from a relevant source position.
- **Unintuitive:** The provided error description is not appointing the error, but the link to the actual error can be discovered based on the provided information. An error is also considered acceptable if it provides an intuitive error message together with an unintuitive hint.
- **Not succinct:** Two or three errors are mentioned by the Ampersand parser.

Good error message: Any error message that is not bad nor acceptable is good.

6.2.2 Gathering process

To gather the necessary input for the as-is analysis, an exhaustive list of all possible error messages is created. This as-is analysis will be used as a reference base to verify the implementation of the new error mechanism with Parsec. The errors are invoked by simulating all possible syntax errors that will invoke an error within the Ampersand parser. Each syntax statement is therefore manipulated, introducing one specific error per time, and the resulting error message is then recorded together with the actual erroneous statement. The exact same statements are afterwards pushed through the new parser, making it possible to make a quantitative 'before and after' analysis. Special attention is given to avoid redundant errors that could influence the quantitative analysis. An example of such an redundant error is the use of a capital letter in defining a specific reference. Although these references are used in several syntax statements, there is only one procedure in the parser to check all references starting with a capital letter. An improvement in the error message of this check may only be taken into account one time.

6.2.3 Results

Based on our error message qualification definition, Table 2 visualizes the results of the as-is analysis. This analysis clearly confirms the statement that the quality of the error messages is of low quality and that there is a lot of room for improvement.

Error quality	Old parser	
Good	19	22,35%
Acceptable	48	56,47%
Bad	18	21,18%
Total	**85**	**100,00%**

Table 2: Error message as-is analysis results

6.3 Scanner

As part of the system analysis, we also analyzed the existing scanner/lexer. The scanner is responsible to split up the input stream into tokens. Tokens are atomic and categorized pieces of information from the input string that can be recognized by the parser.

The following improvement points were identified after the analysis:

Dispersed error messages The error messages produced by the lexer are of good quality. Each error message is, however, defined directly within the corresponding lexer function making the maintenance harder.

Complex token structure The token structure is complex and confusing (the structure is given below). Two values are present in the token, of which one (`val1`) is never used. There is no distinction between the values used to identify the content of the token and the ones to determine the position of the token, i.e. they are in the same data type.

Module structuring In the lexer, the actual lexing functions are intermingled with data types, supporting functions and error message texts. This makes the lexer harder to understand and to maintain.

Language support The errors are returned in English only; no multilingual support is available (or easy to implement).

No support for warnings The lexer can only return errors. Warnings are not supported.

Strings only Token values are stored as strings for all types, with no conversion of values, e.g. integers.

Lacking documentation There was no documentation available on how the lexer was designed and structured.

The tokens from the old parser are represented by the following data type. Note that the comments have been added by us, only in this document.

```haskell
data Token = Tok
 { tp'   :: TokenType -- Identification of the token type
 , val1  :: String    -- This string argument is not used in the lexer.
                      -- For keywords it is filled in but never read.
 , val2  :: String    -- The actual token content, stored as a string
 , pos   :: !Pos      -- Line and column number
 , file  :: !Filename -- File name in which the token is located.
 }

data TokenType
  = TkSymbol    -- ^ Single character symbol
   | TkVarid     -- ^ Lower case identifier
   | TkConid     -- ^ Upper case identifier
   | TkKeyword   -- ^ Keyword
   | TkOp        -- ^ Operator
   | TkString    -- ^ String
   | TkExpl      -- ^ Explanation
   | TkAtom      -- ^ Atom
   | TkChar      -- ^ Single character
   | TkInteger8  -- ^ Octal integer
   | TkInteger10 -- ^ Decimal integer
   | TkInteger16 -- ^ Hexadecimal integer
   | TkTextnm    -- ^ Unused
   | TkTextln    -- ^ Unused
   | TkSpace     -- ^ Unused
   | TkError     -- ^ Unused
  deriving (Eq, Ord)
```

6.4 Grammar

6.4.1 Getting the EBNF in good shape

The Ampersand grammar is described using EBNF notation. EBNF is a notation technique with the goal to express a context free grammar like ADL's. The existing EBNF was outdated and not in line anymore with the actual syntax of Ampersand. As the EBNF is the crucial source of information in building the new parser, the first focus was to update the old EBNF to represent the actual Ampersand Syntax.

Through reverse engineering, we checked all Haskell functions on the actual syntax they implement. In the source of the new parser, all the grammar expressions are placed above the actual parser function as code annotations to support code maintainability.

6.4.2 The actual EBNF diagram

The derived syntax is up to date and visualized using a railroad diagram, an ideal technique to create a visual representation of context free grammars. Several railroad diagram generators are available on the internet, free of charge. We used the railroad diagram generator created by Gunter Rademacher, available on `http://bottlecaps.de/rr/ui`. The generated diagrams with the corresponding EBNF productions are available in the project documentation (appendix).

One interesting outcome is that during the project we found a bug in the Railroad Diagram Generator. The tool would crash with the `Trm4` expressions. This bug was reported to the author Gunther Rademacher, who promptly fixed the issue.

6.5 Parser

The old Ampersand parser was generally well organized, so we mapped each ENBF rule to a different parsing function. However, several flaws were observed as improvement points. We addressed these flaws in our project where possible. The flaws we were not able to solve, due to the constraints of our project, are summarized in Section 9. During this project, we focused on the following issues:

Lacking documentation There was no documentation on the recognized grammar. The last EBNF available was not updated in a long while.

Incompatible parse tree The parser was built with an applicative interface. The applicative operators were thus used in sequence to recognize each of the accepted grammar productions. One of the advantages of the applicative interface is that the code can look very close to the grammar without any 'boiler-plate' code. However, the parser was often forced to change the order and format of the parsed structures, because the parse tree did not match the grammar productions (i.e. many rebuild functions).

Data transformations Normally, a parser is only supposed to recognize a grammar and convert the input to a parse tree. However, in the old parser some transformations and simplifications are executed (e.g. removing double negations). That makes the parser unnecessarily complex and violates the design principle of encapsulation.

Long file Since all the grammar constructions – plus help functions – were in a single file, the parser was hardly readable. It summed a total of 823 lines of code only in the `Parser` module.

Pretty printing It was not possible to print the parse tree back to ADL-code. That made it harder to develop and test the parser properly.

Test suite There were no automated tests for the parser other than the complete chain tests with Sentinel. Because of this, any code change would be hard to test and could potentially influence other Ampersand modules.

Duplicated code A large part of the code was duplicated and not used (mainly in the `Parsing` module).

Error messages As mentioned, the main reason for this project was the bad quality of the error messages generated. Subsection 6.2 expands on this point.

Although it may be hard to resolve all the mentioned issues, we believe our efforts have paid off well. In Subsection 7.4 we describe how the new parser was designed.

6.6 Parse tree

The parse tree (also known as P-structure) is a data structure that very much resembles the EBNF description. The root of the tree is the `P_Context` structure, and every leaf of the tree has a field for the location where it was found in the ADL code (the `Origin` structure). The tree is consistently defined with the record syntax and is well documented.

However, the constructions are not completely pure, since some transformations are necessary from the ADL to the P-structure. This forces the parser to do more than only parsing, as mentioned in Subsection 6.5. Also, the order of the fields can be confusing; sometimes `Origin` is the first field and sometimes it is not.

During this project, small changes to the parse tree have been done. These changes are described in Subsection 7.4.3.

7 Design & Implementation

In this section, we depict the high-level architecture of the new parser. We do this by first giving an architecture overview in Subsection 7.1. Then we elaborate on the code quality in Subsection 7.2. The new lexer is described in Subsection 7.3, while the parser itself is presented in Subsection 7.4. The new test suite and the improved error messages are described later in Section 8. Additionally, more detailed artifacts are given in the project documentation (appendix).

7.1 System overview

The parser module overview is given in Figure 4.

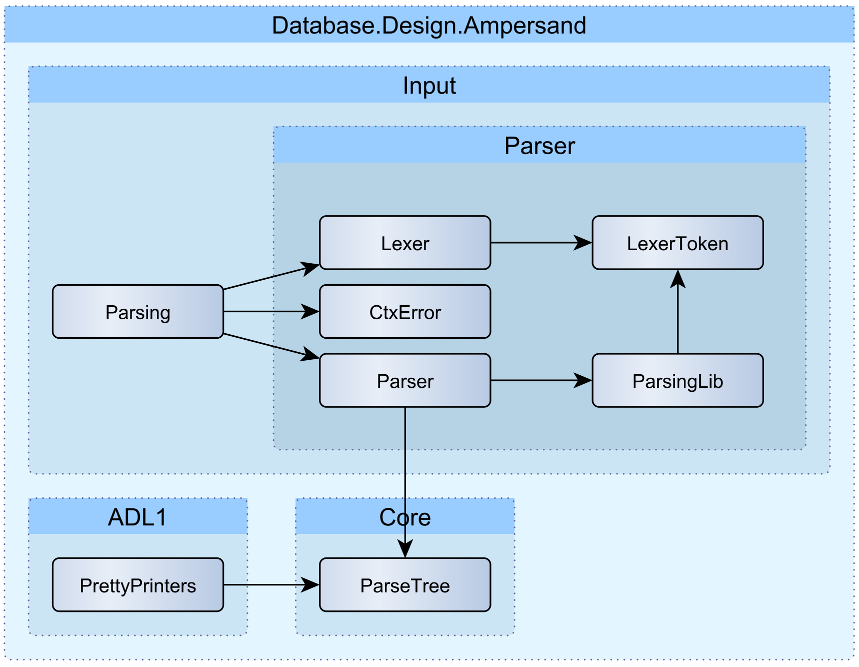

Figure 4: The modules relevant for the parser and their dependencies

The parsing process starts in the module `Parsing`. As a first step, the input string is sliced into tokens by the `Lexer` module. Once the input string is separated into the token structure as defined in the module `LexerToken` the next step is to actually parse the tokens by the `Parser`. The parser will use the `ParsingLib` to create the parse tree as defined in `ParseTree`.

Each main module has the following responsibilities:

Parsing module that implements the interface of the parser with the rest of the system. It is responsible for reading the input files, calling the lexer and the parser and returning a parse tree as result (or a parse error).

Lexer/LexerToken modules responsible for recognizing the input characters and converting them to tokens. The new lexer, together with its sub-modules, splits the input strings into the token structure defined in `LexerToken`. This list of tokens is the actual input for the parser.

Parser module responsible for executing the parsing itself. It accepts the tokens that are allowed in each grammar production and generates the corresponding parse tree. The parser is described in Subsection 7.4.

Several supporting modules are defined and used by one or more main modules:

ParsingLib library that contains several useful functions to assist the parser, e.g. token recognition. These functions are not depending on the specific grammar rules.

ParseTree external module containing the parse tree data structures. Only details of this module have been changed during this project (e.g. field ordering).

PrettyPrinters contains instances of the `Pretty` type class for the parse tree data types and the functions responsible for printing the parse tree to ADL scripts in a 'pretty' way.

CtxError contains the data structures responsible for the parse errors and their location. This module has not been refactored as a part of this project.

7.2 Software quality factors

In our project plan [2] multiple non-functional requirements are included in the project scope. Improving the code maintainability is one of the most important non-functional requirements.

To assure that these non-functional requirements are correctly addressed, we defined some measures to adhere to during the full project life cycle.

7.2.1 Documentation

All important design decisions we made together with the code we delivered need to be documented. This documentation is needed for the Ampersand team to have a clear insight in the way the new parser is structured and how it is integrated in Ampersand. The availability of this documentation is crucial for the maintainability of the new parser. The following documentation is delivered as a result of this ABI project:

System design A general system overview of the new system, describing the goal and purpose of each module, how it is designed to achieve its goals and how it is integrated in the system architecture. The system design is integrated in this thesis document.

Code annotations Haddock is the de-facto standard for generating Haskell documentation. This documentation generator generates HTML based on the comments in the Haskell source code. It is important to remark that Haddock normally only generates documentation of the functions that are exported by each module. It is however important that all functions are well documented, including the internal ones, and therefore, the internal functions are documented using regular, non-Haddock, code annotations. The code annotations, in the source code, together with the Haddock documentation are delivered as an appendix to this document.

EBNF comments The EBNF structure is the most important documentation of how the Ampersand syntax is composed and how the parser functions are defined. As described in Subsection 6.4, the actual EBNF is retrieved through reverse engineering. Each parser function corresponds to a specific EBNF syntax rule and this rule is consistently annotated in the code just above the parse function. A specific markup (`---`) is used to tag the EBNF rules. This allows us to automatically extract the EBNF rules from source code and export them to other formats.

7.2.2 Readability

In the as-is analysis of the current parser, we noticed that the code has been through several feature additions over the past years. These repetitive small changes reflected in parts of the code which has become too elaborated or sometimes even obsolete.

Each code statement in the parser and lexer is analyzed by the project team and, where possible, refactored to be as concise as possible. The delivered code is now as short as possible without compromising the readability of the code.

In addition, the code review tool HLint is used. This tool provides a full overview of several code optimization suggestions that can further optimize the readability of the source code. These suggestions cover topics such as redundant brackets, parameter reductions and shortcut notations. All HLint warnings regarding the input subsystem are fully addressed before the code is delivered to the customer. The HLint report for the delivered code is available in the project documentation (appendix).

7.2.3 Performance

Performance is a requirement often made in software engineering projects that is difficult to measure before the software is actually used in a production environment. For the new Ampersand parser, we proactively identified the topics that could have an impact on the resulting parsing performance. In the design of the new parser, a performance aspect that makes an important difference is the parser backtracking (with the `try` function). Several refactorings in the grammar are carried through to avoid the use of the `try` function. A full list of remaining backtracking productions, together with our suggestions how to solve them, is provided in section Subsection 7.4.2.

7.3 New lexer

7.3.1 The rationale behind the new lexer

In the design of the new Ampersand parser, the first decision to tackle is whether to keep the existing scanner/lexer that is implemented in UULib or to implement a new one with Parsec. In the analysis of the error improvement areas in Section 6, the main improvements are identified within the old parser. The error feedback quality, produced by the scanner module, is higher and therefore, there is no stringent need to re-implement the scanner. On the other hand, given the aspect that Parsec is identified as the new parser library, keeping the current scanner would result in the utilization of two different libraries providing more of less the same functionality.

The alternative of keeping the existing scanner would deliver a perfect functional solution, but mixing these two libraries would increase the complexity of the solution, thus decreasing the maintainability. To avoid this decrease in maintainability, the decision is to implement the parser and scanner based on the same library.

During the implementation of the lexer module, replacing the old scanner, additional attention was given to further improve the quality of the error messages. The scanner module is renamed to lexer to stress the aspect that the principle of lexemes is used in the new scanner. Lexemes can be seen as the part of a token containing the actual language content besides the actual position information.

The lexer is built based on the existing Helium lexer modules. Helium is a Haskell compiler with the main goal of giving user friendly error messages [29]. The lexer module in Helium contains interesting principles such as position monitoring, warnings and easy maintainable error messages.

7.3.2 Lexer structure

The lexer is the main module, in which the actual lexing is done, and to do so, it uses the following sub-modules:

LexerMonad contains a monad definition that supports lexing with context. It tracks for example the location in the input and the warnings that may be generated. This module is based on the Helium lexer, without any modifications to the used functions. All unused functions are removed to improve the code maintainability.

The following functions or types are used in the Ampersand lexer:

- **LexerMonad** is the main monadic type used in the lexer returning an error or a list of tokens together with a list of warnings
- **addPos** is used to trace the position of the token
- **lexerError** to generate lexer error
- **lexerWarning** to generate lexer warnings
- **runLexerMonad** main function to handle the `LexerMonad` results

LexerMessage contains functions to handle errors and warnings from the lexer. Based on the warning/error type and the needed language, `LexerMessage` will fetch the correct description of an error or a warning out of the `LexerTexts` module. The show functions for the error and warning are maintained in this module.

LexerTexts fetches the correct description of an error or a warning out of the `LexerTexts` module. The centralization of the error message texts provides an easy entry point for the maintenance of the actual messages as these messages are no longer dispersed over the module functions.

LexerBinaryTrees module responsible for searching binary trees in an efficient way, to support the token recognition. This is the previously existing `UU_BinaryTrees` module which is renamed to match the used naming structure of the new lexer modules.

LexerToken contains the data structure and corresponding show function that represents the input tokens for the lexer.

7.3.3 New token structure

Based on the improvement topics mentioned in Subsection 6.3, a new token structure is defined. Each token contains the lexeme: a part of the input string defining the token type and content, plus the position of the token in the input file. The token structure is defined as follows:

```haskell
data Token = Tok { tokLex :: Lexeme     -- ^ The lexeme (defined below)
                 , tokPos :: FilePos    -- ^ The file position
                 }

data Lexeme   = LexSymbol      Char     -- ^ Single character symbol
              | LexOperator    String   -- ^ Operator
              | LexKeyword     String   -- ^ Keyword
              | LexString      String   -- ^ String
              | LexExpl        String   -- ^ Explanation
              | LexAtom        String   -- ^ Atom
```

```
                | LexDecimal     Int       -- ^ Decimal integer
                | LexOctal       Int       -- ^ Octal integer
                | LexHex         Int       -- ^ Hexadecimal integer
                | LexConId       String    -- ^ Upper case identifier
                | LexVarId       String    -- ^ Lower case identifier
  deriving (Eq, Ord)
```

`Lexeme` is the combination of the token type and the actual token content, sliced from the input string. `FilePos` is used to keep track of the original position of the lexeme in the input string.

During the lexer processing, the input file is processed sequentially. All kinds of different accepted constructions are checked in a specific order. Each time a match is found, the lexeme is extracted from the input string and a token is created. In the token creation (function `returnToken`), the position and the lexeme are grouped into a token, then the next lexer iteration is started.

7.4 New parser

The high-level design of the new parser has not changed much. While the new parser may still be recognizable for the Ampersand developers, several improvements have been made.

As decided during the research for domain and techniques (see Section 4), the parser has been rebuilt with the Parsec combinator library. Basically, each EBNF rule receives its own parser function. Thanks to the combinator operators, each parsing function also looks very similar to its corresponding EBNF.

The applicative interface is consistently used. By changing details of the implementation, e.g. the order of the fields in the parse tree, we have made many of the 'rebuild' functions unnecessary. For some parsers the amount of changes necessary in order to remove supporting functions was too large or even impossible with the current parse tree.

Note that in parts of the parser, the function syntax has substituted the record syntax for creating data objects. This was done only when the code readability could be improved by doing so.

7.4.1 Parsec

As mentioned in Section 4, the new Ampersand parser has been rebuilt with another parsing library, namely Parsec. However, for the Ampersand developers, the source code of the parser will still look very familiar, thanks to the applicative interface. For developers, the main differences between Parsec and the UUlib are:

- Parsec does not backtrack by default. In order to enable backtracking, the `try` function must be used. This is described in Subsection 7.4.2.
- Parsec does not try to solve parsing errors. The parser stops immediately after the first issue. This way, the user is not overwhelmed with irrelevant information. See also the error analysis in Subsection 8.2.
- Error messages are customizable by using the `<?>` operator. This is also suggested in Section 9.
- Some combinators have a different name, e.g. one must use `option` instead of `opt`. Because the documentation found on Hackage is clear and sufficient, interface differences are not documented here.

7.4.2 Backtracking

In order to explain the differences on backtracking behavior between the UUlib and Parsec, we quote here Doaitse Swierstra, the author of the UUlib [52]:

> *To understand the subtleties it is important to understand the differences between the try construct in Haskell and the non-greedy parsing strategy used in UU-parsinglib. Effectively the latter is a try which just looks ahead one symbol. In that respect it is less powerful than the `try` construct from Parsec, in which you specify that a specific construct has to be present completely. And then there is the underlying different overall strategy. Parsec uses a back-tracking strategy with explicit tries to commit, whereas UU-parsinglib uses a breadth-first strategy with an occasional single symbol look-ahead.*

Although the `try` construct for backtracking in Parsec is very powerful, it is also undesirable: Backtracking increases the parser's memory usage, speed, maintainability and the quality of the error messages [32]. However, they are necessary when the grammar is not left-factored [32]. In this section we explain why each of the remaining try statements are necessary, and how these issues can be resolved:

Classify This ambiguity in the grammar arises from the `Classify` and `GenDef` productions:

```
Classify ::= 'CLASSIFY' ConceptRef 'IS' Cterm
GenDef ::= ('CLASSIFY' | 'SPEC') ConceptRef 'ISA' ConceptRef
```

When the parser encounters `'CLASSIFY'`, the parser cannot determine whether it found a `Classify` or a `GenDef` production. Therefore, the parser must consume the keyword and a `ConceptRef` before consuming either `'IS'` or `'ISA'` and determining which production is applicable.

In order to solve this issue, one must choose a different keyword or symbol for each of the productions. Another option would be to merge the two statements in the same parser. We did not merge the productions because that would make the parser less maintainable.

Role This ambiguity in the grammar arises from the `RoleRelation` and `RoleRule` productions:

```
RoleRelation ::= 'ROLE' RoleList 'EDITS' NamedRelList
RoleRule ::= 'ROLE' RoleList 'MAINTAINS' ADLidList
```

When the parser encounters `'ROLE'`, it cannot determine whether it is a `RoleRelation` or a `RoleRule` production. Therefore, the parser must consume the keyword and a `RoleList` (which may be long) before consuming either `'MAINTAINS'` or `'EDITS'` and determining which production is applicable.

In order to solve this issue, one must choose a different keyword for each of the productions, merge the two options to have the same representation in the parse tree, or refactor the parser so that the two options are parsed together. We did not merge the productions because that would make the parser less maintainable.

View This ambiguity in the grammar arises from the `FancyViewDef` and `ViewDefLegacy` productions:

```
  FancyViewDef ::= 'VIEW' Label ConceptOneRefPos 'DEFAULT'?
'{' ViewObjList '}' HtmlView? 'ENDVIEW'
  ViewDefLegacy ::= ('VIEW' | 'KEY') LabelProps
ConceptOneRefPos '(' ViewSegmentList ')'
```

When the parser encounters ‘VIEW’, it cannot define whether it found a FancyViewDef or a ViewDefLegacy production. In this case, defining which construction is applicable is even more complicated. This decision must, in the worst case, be delayed until the parser encounters a ‘{’ or ’(’. That is because the productions Label and LabelProps are not disjoint, and ‘DEFAULT’ is optional.

In order to solve this issue, we advise to merge or drop the legacy statement.

Multiplicity This ambiguity in the grammar arises from the Mult production:

```
    Mult ::= (‘0’ | ‘1’) ‘..’ (‘1’ | ‘*’) | ‘*’ | ‘1’
```

When the parser encounters ‘1’, it cannot define whether it found the first or the last production. The parser must therefore read the next token before choosing the right option.

In order to solve this issue, we advise to refactor the grammar (and the parser) to have the following production:

```
    Mult ::= ‘0’ ‘..’ (‘1’ | ‘*’) | ‘1’(‘..’ (‘1’ | ‘*’))? | ‘*’
```

We did not refactor the code in this manner because the pMult parser does more than only parsing: it also changes the representation of the found constructions before creating the parse tree.

Labels and Terms In two of the productions of the grammar, an ambiguity arises when an optional Label production is followed by a Term production (Label? Term). The issue is that Label, LabelProps, Rule and Term may all begin with a Varid:

```
    Label ::= ADLid ‘:’ => Varid ‘:’
    LabelProps ::= ADLid (‘{’ADLidListList‘}’)? ‘:’ => Varid ‘:’
    Rule ::= Term (‘=’ Term | ‘|-’ Term)? => Term
    Term => Trm2 => Trm3 => Trm4 => Trm5 => Trm6 => RelationRef
  => NamedRel => Varid Sign?
```

This ambiguity exists in the Att and RuleDef productions:

```
    Att ::= LabelProps? Term
    RuleDef ::= ‘RULE’ Label? Rule Meaning* Message* Violation?
```

What happens here is that when the parser encounters a Varid, it cannot define whether it is part of the (optional) Label production or if no Label was given and the Varid is part of a Term/NamedRel production.

Due to the quite complex grammar for the Term production, this issue may severely impact the parser’s performance. This is probably the most harmful of the ambiguities mentioned. However, it can only be solved by adding a symbol before the Term production (e.g. making the ‘:’ non-optional).

Please note that in order to have proper backtracking with correct error messages, Parsec may require two try-statements [51].

7.4.3 Changes to the parse tree

Improvements in the Ampersand parse tree are out of the scope of this project, because of the potential consequences to the rest of the Ampersand system. However, during the development of the new parser a few constructions have been changed in order to make the parser more readable and maintainable. The changes have been mostly in the order of the constructor parameters, and this was done consequently though all Ampersand modules. The updated parse tree is depicted in the project documentation (appendix).

8 Test Report

8.1 Test suite

Together with the new parser, a test suite has been developed. This test suite has been used to verify the performance and correctness of the new parser. The source code can be found in the directory `src/Database/Design/Ampersand/Test` within the Ampersand repository.

The test suite runs in three steps (see Figure 5). The first step is to check if a list of input files can be parsed successfully. If no issues are found during the parsing, the same list is used by the module `RunAmpersand` in which all input scripts are processed completely by Ampersand. As a final step, a series of random generated parse tree structures is generated, translated in ADL files and tested. Each of the modules is described in the following subsection.

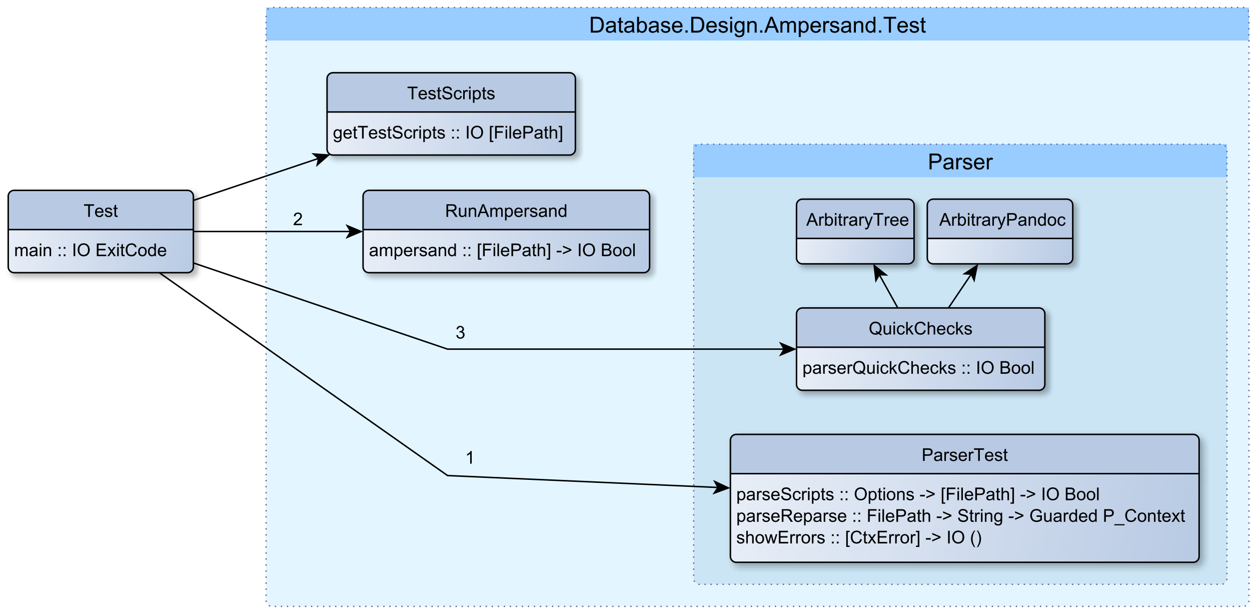
Database.Design.Ampersand.Test
TestScripts
getTestScripts :: IO [FilePath]
Test
main :: IO ExitCode
2
RunAmpersand
ampersand :: [FilePath] -> IO Bool
Parser
ArbitraryTree
ArbitraryPandoc
3
QuickChecks
parserQuickChecks :: IO Bool
1
ParserTest
parseScripts :: Options -> [FilePath] -> IO Bool
parseReparse :: FilePath -> String -> Guarded P_Context
showErrors :: [CtxError] -> IO ()

Figure 5: Test suite modules with their exported functions

8.1.1 Modules

In this section a short description of each module is given:

Test contains the `main` method that can be executed to run the test suite. The `main` function calls each of the other modules in sequence, stopping if any of them returns `False`. When all tests have been successful, the return code is `ExitSuccess`. Otherwise, the return code is naturally `ExitFailure`.

ParserTest exports three functions that are the core of testing the parser:

- `parseScripts` receives a list of files to parse, and checks that every file can be parsed successfully.
- `parseReparse` tries to parse a file, and if successful, pretty-prints the result and parses it again.
- `showErrors` prints the given parse errors to the output.

RunAmpersand receives a list of files, and checks that every file can be executed successfully by Ampersand. This tests thus not only the parser, but also the interface between the parser and the type checker, as well as the rest of the Ampersand chain.

The following modules are supporting one ore more of the three main modules:

TestScripts retrieves a list of scripts that can be used for the different tests. It searches for tests within the directory `ArchitectureAndDesign`, and contains a list of scripts from the `ampersand-models` repository, that can be changed at a later moment if wished. Note that all the ADL-scripts listed in this section must be correct for the parser and the type checker. During development, the list was limited to the scripts that could be successfully executed by the original Ampersand version.

QuickChecks generates random parse tree structures and generates the corresponding ADL-script by pretty printing the parse tree. This ADL-script is then fed back to the parser through the `parseReparse` function, to verify that the parser can accept any random input. More information on the quick checks is given in subsection 8.1.2.

ArbitraryTree is a support module that gives `Arbitrary` instances to all parse-tree structures. This is used by QuickCheck as described in subsection 8.1.2.

ArbitraryPandoc contains `Arbitrary` instances to the Pandoc data types. This file has not been developed in this project, but copied from the `jgm/pandoc` project with the GPL license.

8.1.2 QuickCheck and pretty printing

The most innovative part of the test suite is the use of random structures to test the parser. In this section we describe how this generation is implemented.

The main role in the generation of random structures is played by the support library QuickCheck [37], which has been added to the Ampersand project. QuickCheck is able to generate any data structure randomly. However, since the parse tree is a custom structure that must obey specific rules, QuickCheck requires the specification of these rules by instances of the `Arbitrary` class.

Every data structure in the parse tree has received an `Arbitrary` instance used for test purposes. The instances can be found in the module `ArbitraryTree`, as described in subsection 8.1.1.

After generating the random parse trees, the test suite needs to convert them to ADL-scripts. The conversion of parse tree to source code is also known as pretty printing. As the pretty printing is seen as part of the parse tree, it is not included in the Test modules, but is part of the input subsystem. The pretty printing instances are found in the module `Database.Design.Ampersand.ADL1.PrettyPrinters`. This module makes use of the library `Text.PrettyPrint.Leijen`, that outlines the output so it is indeed ‘pretty’.

Now that the ADL source is available, the parser is executed. The result of the parser is checked to be equal to the generated tree by the property `prop_pretty`. The property is currently configured to be tested for 64 random parse trees in each run. If the test fails for any generated structure, the test suite fails with an appropriate error.

8.1.3 Running the tests

During the parser development, the `main` function of the parser tests has been executed manually, through a batch file. This is mainly done because the project team did not have access to the Sentinel server, and no documentation was available on how to run Sentinel locally on a Windows machine. However, now that the parser is being delivered, it should be integrated with the other existing Ampersand/Sentinel tests. We leave the option open for the Ampersand development team to either add the Sentinel jobs to this test suite, or to add the parser test suite to the Sentinel jobs.

8.1.4 Test coverage

The main objective of the test suite is naturally to test the parser. By using HPC (Haskell Program Coverage) we verified that the most important parts of the code were well tested. For instance, the `Parser` module is covered by 96%. `ParsingLib` is 87% covered, while the module `Lexer` is 82% tested. Finally, the module `PrettyPrinters` is 100% covered. The complete list with the code coverage is available in the generated HPC report, which is part of the project documentation (appendix).

Note that the parts of the code that were not tested, could not be tested for two reasons: only valid ADL files are tested automatically; the incorrect files are tested manually (see Subsection 8.2). The other reason is that the ADL files provided did not contain all possible grammar constructions. Given the extend to which the errors are tested manually we believe that the full test coverage will be very close to 100%.

8.2 Error messages

The proof of the pudding is in the eating, and therefore, our error message qualification definition from Subsection 6.2 is applied to the new parser. All erroneous syntax statements recorded during the analysis phase are fed to the new parser to draw up the error message quality overview of the new parser.

The results are placed in Table 3 next to the results of the old parser as a reference point. The actual error list, containing the syntax statements with the corresponding error messages, is available in the project documentation (appendix).

Error quality	**Old parser**		**New parser**	
Good	19	22,35%	70	82,35%
Acceptable	48	56,47%	14	16,47%
Bad	18	21,18%	1	1,18%
Total	**85**	**100%**	**85**	**100%**

Table 3: Error message comparison results

The table clearly visualizes that there indeed was an issue with the error messages generated by the old Ampersand parser. By implementing the new parser, a substantial improvement is made towards the creation of good error messages in which 82% of the generated error messages are good, compared to 22% in the old parser.

One bad error message remains in the system. After analysis, the resolution of this remaining bad error would require too much effort, hence increased complexity, compared to the actual value gain. The error in question is given when the `Meta` statement is used with only one string value. Given the simple structure of the notation, we assume that the reference to the line number of the `Meta` is sufficient although the error message is unclear. Therefore, this message is kept as is.

A typical example of an error message improvement is presented below.

The following syntax error is reported as an error by both the old and the new parser:

```
CONTEXT DeliverySimple INCLUDE 0FILEPATH IN ENGLISH
```

The main issue in the example is that the requirement engineer misplaced the language reference as this had to be mentioned before the include statement. Another minor error is that the include statement should be a quoted string. This error is indicated by the new parser in the following way:

```
"ArchitectureAndDesign/Syntax/testfile_mba.adl" (line 1, column 24):
unexpected Keyword "INCLUDE"
expecting Keyword "IN"
```

This error message is concise, and provides a good indication of the error position and the expected input. The old parser reported the same syntax error in a very elaborated and indistinct way:

```
Error(s) found:

before "0" at line 1, column 32 of file "testfile_mba.adl"
Expecting string ""
Try deleting symbol "0" at line 1, column 32 of file
"testfile_mba.adl"

================================

before upper case identifier FILEPATH at line 1, column 33 of file
"testfile_mba.adl"
Expecting string ""
Try inserting symbol string ""

================================

before upper case identifier FILEPATH at line 1, column 33 of file
"testfile_mba.adl"
Expecting lower case identifier ?lc? or "CLASSIFY" or "CONCEPT" or
"ENDCONTEXT" or "IDENT" or "INCLUDE" or "INTERFACE" or "KEY" or
"META" or "PATTERN" or "PHPPL␣UG" or "POPULATION" or "PROCESS" or
"PURPOSE" or "RELATION" or "RULE" or "SPEC" or "SQLPLUG" or
"THEMES" or "VIEW"
Try inserting symbol "PURPOSE"

================================

before upper case identifier FILEPATH at line 1, column 33 of file
"testfile_mba.adl"
Expecting "CONCEPT" or "CONTEXT" or "IDENT" or "INTERFACE" or
"PATTERN" or "PROCESS" or "RELATION" or "RULE" or "VIEW"
Try inserting symbol "CONCEPT"

```

```
================================

before lower case identifier clientName at line 14, column 1 of
file "testfile_mba.adl"
Expecting "HTML" or "LATEX" or "MARKDOWN" or "REF" or "REST" or
explanation {+-}

Try inserting symbol explanation {+-}
```

General remarks and highlights:

- If the error qualification was defined in a less strict way, the numbers would be slightly different as some bad errors would be judged as being acceptable. In the comparison between the old and new parser, the exact same qualification method is used while the syntax issue was, as well, identical. Due to this approach, we can conclude that the achieved results are justified.
- Error positioning was already correct in the old Ampersand parser; this is maintained in the new parser.
- The principle of Parsec to end parsing after the first error is found to have a massive improvement on the bad error messages due the conciseness issues. Nearly 50% of the acceptable and bad messages changed to good error messages just due to this principle.
- Some error messages were extremely elaborated due to which the standard Windows command prompt was unable to fully show the error message.

To provide a better insight in the full list of the old and new error messages, together with the error qualification is provided in the project documentation (appendix).

Based on the defined error qualification method, we can clearly conclude that the implementation of the new Ampersand parser has a very positive effect on the error messages quality. The percentage of good errors is now on an acceptable level and the remaining errors are, except for one, of acceptable quality, while still giving valuable information that can be used for error resolution.

To avoid the risk of additional complexity in the parser code, thus decreasing the code maintainability, we judged that this result is the best compromise between error message quality and maintainability.

9 Recommendations

Although we believe the new parser is a large step forward, we also recognize that there are still improvements to be made.

Ampersand is growing and changing in a fast pace. This is a direct consequence of a project involved with research and active development. In such projects, it is often impossible to predict which functionalities will be necessary and to design the domain-specific language accordingly beforehand.

As expected we see that most of the remaining issues are related to the grammar ambiguities that force backtracking. Also the parse tree is not consistently designed. These issues are mentioned in Subsection 9.1. Other improvements are also possible in the test framework delivered, which are mentioned in Subsection 9.2. Finally, we missed more support from the Ampersand website and/or wiki. This is explained in Subsection 9.3.

9.1 Design

During the parser reimplementation and code refactoring to enhance the code maintainability, we noticed potential improvement topics in Ampersand. Where possible, with respect to our project requirements and our milestones, we integrated these topics in the new parser. Some topics we could not handle due to time constraints or given the undesired impact on the surrounding modules. These improvement ideas can help the Ampersand team to further enhance the tool and therefore, these topics are listed in this section. This section summarizes our improvement suggestions, both generic as specific ones, for the Ampersand tool.

Syntax improvements In the syntax, we discovered some statements which are not pure LL(1) statements. The Parsec `try` function allows the backtracking in the parser, avoiding that input is consumed which is still needed in other parse statements if the parse function cannot succeed successfully. Using `try` we can handle this situation, but the backtracking has a negative impact of the parser performance whilst it adds complexity to the parser module. Our suggestion is to further optimize the Ampersand syntax to establish a pure predictive syntax in which no backtracking is needed. This will not only improve the parser performance and maintainability, the syntax simplifications may also make it easier for new users to learn the language. The syntax statements in which backtracking is needed, including the reasons why and how to solve them, are listed in section 7.4.2.

Warnings An improvement point in the new lexer is that warnings are now supported. Warnings are, however, not yet integrated in the Ampersand tool. There is no need to stop the compiling process for warnings, still the reasons behind them can make sense. Warnings can, however, support the user identifying the cause of unexpected results, although the compilation could be completed successfully. Some examples of warnings are the notification of nested comments or wrong float number notations. Our suggestion is to add a list of warnings to the design artifacts of Ampersand, available for the user to reflect on.

Uniform parse tree structure In the parse tree, not all data types are constructed in correspondence to the syntax of the Ampersand language. Several restructuring functions are used in the parser to reformulate the result of the parse functions to match the constructor type in the parse tree. These functions can be recognized in the parser as rebuild or reorder functions. An example of such a rebuild action is given below: The syntax notation of `ViewAtt` is defined as:

```
  ViewAtt ::= LabelProps? Term
```

The parser function will use the order of the notation to extract the needed information:

```
  pViewAtt :: AmpParser P_ObjectDef
  pViewAtt = rebuild <$> currPos <*> optLabelProps <*> pTerm
      where rebuild pos (nm, strs) ctx =
        P_Obj nm pos ctx mView msub strs
        mView = Nothing
        msub  = Nothing

```

When we look at the data type as defined in the parse tree, the following order is defined:

```
data P_ObjDef a =
   P_Obj { obj_nm :: String
         , obj_pos :: Origin
         , obj_ctx :: Term a
         , obj_mView :: Maybe String
         , obj_msub :: Maybe (P_SubIfc a)
         , obj_strs :: [[String]]
         }

```

As the order of the parameters in the parse tree and the parser are different, the local function `rebuild` is used to align both types to each other. Our personal experience was that these rebuild functions make the parser code more difficult to understand and hence, to maintain. During the project, we have reordered some of the structures of the parse tree. However, we could not do it for all structures. To further decrease the code complexity, we suggest to try to eliminate these rebuild functions by restructuring data types in the parse tree.

Manual overrule of error message Our analysis of the new error messages showed that the quality of these improved distinctively. Parsec provides the possibility to overrule the standard Parsec error message by an own formulated message. During our implementation we decided to stick to the standard if the standard error message was at least sufficient. If the Ampersand teams want to tweak error messages after all, this can be realized using the `<?>` Parsec operator. Placing the `<?>` after the parser, followed by a string, changes the standard Parsec text after the word 'expecting'.

Smaller improvements In the actual parser code, some smaller improvement topics are identified with comments. However, resolving these independent minor topics impacts the parse tree and therefore, these modifications are still open. All improvement topics are documented in the code itself.

9.2 Test suite

In this section we name a couple of changes that can be done to improve the test suite in the future:

Sentinel During the development of the new parser, we worked in a separate fork. Our changes were not being tested in the Ampersand test server (Sentinel). Since we did not have access to this server, we developed a separate test suite. It may be better to integrate the Sentinel jobs into the test suite or to integrate the test suite into the Sentinel jobs.

Output Currently, part of the test suite outputs are written by using the `Debug.Trace` module. From a purely functional perspective, using this module may be undesirable. Therefore, the Ampersand team may consider changing the test outputs to use IO with monads in a more functional way.

9.3 Website and wiki

During the initiation phase of the project we tried to gather as much information as possible regarding the Ampersand project. The Ampersand project currently has two information sources, a Wiki page and a GitHub page. The Wiki page focuses on the goals and the practical usage of Ampersand where the GitHub page is more oriented to the Ampersand developers.

We experienced that it was not always easy to find the information we were looking for on these sites. Some information was outdated and sometimes links were broken. Our suggestion is to have both sites more aligned to each other and to take some time to refresh the current Wiki pages. This will make it easier for new Ampersand users and developers to have a kick-start in the Ampersand world. It may also make it more attractive for open source developers to support the project.

10 Conclusion

Implementing a new parser for Ampersand is a challenge in which different aspects need to be considered. Ampersand has the objective to provide an answer to known issues related to formal techniques used to define software requirements. By using natural language in combination with relational algebra to define business rules, the rules are presented in a way that can be understood by business analysts while guaranteeing consistency. The keystone of the methodology is the supporting tool, which generates useful design artifacts and working prototypes.

To improve the user feedback quality of the Ampersand parser, a combinator library is preferred instead of a parser generator, since the way the combinators work are easier to understand and the generated errors are more customizable. Out of several parsing libraries, the Parsec library had the best references and support documentation and is therefore the library of choice for the Ampersand project.

Based on a error manifesto for defining good and bad error messages, the initial measurement of the old parser clearly envisioned the need for improvement. In addition, due to the evolutionary growth of the parser code, simplification was a necessity. To achieve this, several tools are used, a new test suite is developed and good documentation is written. Grammar optimizations to avoid the need for backtracking and data restructuring to align the parse tree on the grammar have also improved the code maintainability.

For the future, several additional improvement areas outside the scope of this project are identified. These topics include the introduction of warnings, the further grammar optimization to a pure predictive parser and additional optimizations towards the parse tree.

Based on the error result analysis as described in Subsection 8.2 and the overall code maintainability actions, we believe we succeeded in the realization of our project goals. It must be said that it will take some time before the actual benefits will be visible, since it takes some time before the Ampersand users can be consulted about the improved user feedback from the parser.

One of the criteria to further stimulate and promote the introduction of Ampersand in both commercial and educational contexts is the availability of a large user base. We are convinced that this project will improve the customer satisfaction, which on its turn will have a positive feedback on the user base extension.

Appendices

Project documentation

All the necessary information for the evaluation and usage of the new Ampersand parser that is not included in this thesis has been delivered separately in the project documentation. The documentation contains the following artifacts:

Code A copy of the code itself, as delivered through GitHub.

Diagrams Several design diagrams with the parse tree and Haskell modules.

Ebnf The EBNF grammar of ADL in text and interactive diagram format.

Errors The complete list with the comparison between the old and the new parser and the source code for each tested error.

HLint Report with the static code analysis of HLint.

Hpc Haskell program coverage for the test suite.

Haddock The code documentation exported to HTML with the Haddock tool.

Notepad++ XML file that can be used to display ADL code in Notepad++.

Glossary

A-structure The ADL code generated by the Ampersand type checker, used as input for the calculator component. 10

ABI Afstudeerproject Bachelor Informatica (graduation project for the computer science bachelor). 35

ADL Ampersand Design Language. 9

ADL-structure See A-structure. 10

Alex Lexer included in the Haskell Platform. 20

BNF Backus-Naur Form, a meta-syntax notation for expressing context-free grammars. 32

CML Conceptual Modeling Language. 13

COTS Commercial Off The Shelf. 13

DSL Domain specific language. 20

EBNF Extended Backus-Naur Form, an extension on BNF. 32

EDSL Embedded Domain Specific Language. 20

Extended Backus-Naur Form Notation technique for documenting context-free grammars. 32

F-structure The functional structure generated by the Ampersand calculator, used as input for the different output modules. 10

GADT Generalized Algebraic Datatypes for Haskell. 22

GCC Gnu Compiler Collection. `https://gcc.gnu.org/`. 21

GHC Glasgow Haskell Compiler. 21

Haddock Software documentation generator for the Haskell programming language. 11

Happy Parser Generator Parser generator system for Haskell. `https://www.haskell.org/happy/`. 21

Hellium Haskell Compiler. `www.cs.uu.nl/helium`. 21

HLint Statical analysis software that suggests maintainability improvements. 11

HPC Haskell Program Coverage. 11

Hugs Haskell Compiler. 21

i* i star. 13

KAOS Knowledge Acquisition in autOmated Specification. 13

LALR parser LR parser with look-ahead. 21

Larch Languages and Tools for Formal Specification. 12

Lexer Software that does the lexical analysis. 20

Lexical analysis Separating text into tokens. 20

LL(k)-grammar A grammar that can be parsed by an LL(k)-parser. 21

LL(k)-parser Top-down parser that parses from left to right, performing the leftmost derivation with a maximum k-tokens of look-ahead. 21

LOTOS Language Of Temporal Ordering Specification. 12

LR parser Top-down parser that parses from left to right, performing the rightmost derivation. 21

P-structure The parse-tree generated by the Ampersand parser, used as input for the type checker. 10

Parsec Haskell monadic parsing combinator library written by Daan Leijen. `https://www.haskell.org/haskellwiki/Parsec`. 23

RAISE Rigorous Approach to Industrial Software Engineering. 12

RML Requirement Modeling Language. 13

Telos From the Greek word which means end; the object aimed at in an effort; purpose. 13

UU-parsinglib New version of the UUlib. 22

UUlib Haskell parsing library from the Utrecht University. `http://foswiki.cs.uu.nl/foswiki/HUT/ParserCombinators`. 22

VDM Vienna Development Method. 12

YACC Yet Another Compiler Compiler. 21

Z Formal specification notation zed. 12

References

Project references

[1] Bastiaan Heeren and Stef Joosten. Projectaanvraag Afstudeerproject bachelor informatica – ABI. Bruikbare feedback in de Ampersand parser, 2013.

[2] Maarten Baertsoen and Daniel S. C. Schiavini. Planning for the project 'Useful feedback in the Ampersand parser'. Version 2.0, 2014.

[3] Maarten Baertsoen and Daniel S. C. Schiavini. Research context. Version 1.0, 2015.

Academic references

[4] Stef Joosten. Bedrijfsregels: zin, onzin en een perspectief. *Informatie magazine*, January/February 2010:44–49, 2010.

[5] Stef Joosten, Rieks Joosten, and Sebastiaan Joosten. Ampersand: foutvrije specificaties voor B&I-vraagstukken. *Informatie magazine*, August 2007:42–50, 2007.

[6] Daan Leijen. Parsec, a fast combinator parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), October 2001.

[7] Pim Bos. Bedrijfsregels in verschillende vormen – een vergelijking op toepasbaarheid tussen SWRL en relatie algebra bij wetteksten. Master's thesis, Open Universiteit Nederland, 2013.

[8] Ronald G. Ross. Business problems addressed by the business rule approach. *Business Rules Journal*, 4(3), 2003.

[9] Thomas E. Bell and T. A. Thayer. Software requirements: Are they really a problem? In *Proceedings of the 2nd International Conference on Software Engineering*, pages 61–68, 1976.

[10] Ivan J Jureta, Alex Borgida, Neil A Ernst, and John Mylopoulos. Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling. In *2010 18th IEEE International Requirements Engineering Conference*, pages 115–124. IEEE, 2010.

[11] Luqi and J. A. Goguen. Formal methods: Promises and problems. *IEEE Software*, pages 73–85, 1997.

[12] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28:626–643, 1996.

[13] J. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 1989.

[14] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11, 2002.

[15] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14:25–59, 1987.

[16] Chris George and Anne E. Haxthausen. The logic of the raise specification language. *Computing and Informatics*, 22, 2003.

[17] Razvan Diaconescu, Kokichi Futatsugi, and Kazuhiro Ogata. Cafeobj: Logical foundations and methodologies. *Computers and Artificial Intelligence*, 22:257–283, 2003.

[18] Eric S. K. Yu and John Mylopoulos. Understanding 'why' in software process modelling, analysis, and design. In *International Conference on Software Engineering*, pages 159–168. IEEE Computer Society Press, 1994.

[19] Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: RML revisited. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 135–147. IEEE Computer Society Press, 1994.

[20] John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *Information Systems*, 8:325–362, 1990.

[21] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[22] Eric S. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *International Symposium on Requirements Engineering*, pages 226–235, 1997.

[23] Ivan Jureta, John Mylopoulos, and Stéphane Faulkner. Revisiting the core ontology and problem in requirements engineering., 2008.

[24] S. Doaitse Swierstra. Combinator parsing: A short tutorial. Technical report, Utrecht University, 2009.

[25] Arie Middelkoop, Atze Dijkstra, and S Doaitse Swierstra. Dependently typed attribute grammars. In *Implementation and Application of Functional Languages*, pages 105–120. Springer, 2011.

[26] Graham Hutton and Erik Meijer. Functional pearls – monadic parsing in Haskell. Technical report, University of Nottingham, 1996.

[27] Robert C. Moore. Removing left recursion from context-free grammars. In *6th Applied Natural Language Processing Conference: 249–255*, 2000.

[28] Paul Purdom. The size of LALR(1) parsers. *BIT Numerical Mathematics*, 14(3):326–337, 1974.

[29] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71. ACM, 2003.

[30] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. *Advanced Functional Programming, Lecture Notes in Computer Science, Springer, Berlin*, 1129:184–207, 1996.

[31] Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming 18*, page 1–13, 2008.

[32] Daan Leijen. Parsec: Direct style monadic parser combinators for the real world. Technical report, University of Utrecht, 2001.

[33] Bastiaan Heeren. *Top Quality Type Error Messages.* PhD thesis, Utrecht University, 2005.

[34] Michael Spenke, Heinz Muhlenbein, Monika Mevenkamp, Friedemann Mattern, and Christian Beilken. A language independent error recovery method for LL(1) parsers. *Software Practice and Experience, Vol. 14(11) 1095-1107*, 1984.

[35] Jun Yang. *Improving polymorphic type explanations.* PhD thesis, Heriot-Watt University, 2001.

[36] Jun Yang, Greg Michaelson, Phil Trinder, and J.B. Wells. Improved type error reporting. In M. Mohnen and P. Koopman, editors, *In Proceedings of 12th International Workshop on Implementation of Functional Languages*, volume 2011 of LNCS, pages 71–86. RWTH Aachen, Springer Verlag, September 2000.

[37] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *ACM sigplan notices*, 46(4):53–64, 2011.

Other references

[38] Stef Joosten. Deriving functional specifications from business requirements with Ampersand, 2007. `http://icommas.ou.nl/wikiowi/images/e/e0/ampersand_draft_2007nov.pdf`.

[39] Ronald G. Ross. The business rules manifesto, 2003. Version 2.0, `http://www.businessrulesgroup.org/brmanifesto.htm`.

[40] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9 edition, 2010.

[41] Andreas Müller. VDM – The Vienna Development Method. *Bachelor thesis in "Formal Methods in Software Engineering", Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria*, 2009.

[42] Joseph A Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. *Introducing OBJ*. Springer, 2000.

[43] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and tools for formal specification*. Springer-Verlag, 1993.

[44] Frederico de Oliveira Jr., Ricardo Lima, Marcio Cornelio, Sergio Soares, Paulo Maciel, Raimundo Barreto, Meuse Oliveira Jr., and Eduardo Tavares. Cml: C modeling language, 2007. `http://www.jucs.org/jucs_13_6/cml_c_modeling_language`.

[45] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot, 2006. `http://www.graphviz.org/Documentation/dotguide.pdf`.

[46] Lars Marius Garshol. BNF and EBNF: What are they and how do they work?, 2008. `http://www.garshol.priv.no/download/text/bnf.html`.

[47] Geoff Hulette. Haskell parser examples, 2014. `https://github.com/ghulette/haskell-parser-examples`.

[48] Nikolaos Bezirgiannis. Benchmarking parsers, 2012. `http://bezirg.net/afp_presentation.pdf`.

[49] S. Doaitse Swierstra. Haskell mailing list, 2009. `https://www.mail-archive.com/haskell@haskell.org/msg22001.html`.

[50] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, 2008. Chapter 16, Using Parsec. `http://book.realworldhaskell.org/read/using-parsec.html`.

[51] Edward Z. Yang. Parsec: "try a <|> b" considered harmful, 2014. `http://blog.ezyang.com/2014/05/parsec-try-a-or-b-considered-harmful/`.

[52] Doaitse Swierstra. Performance of UU-parsinglib compared to "try" in Parsec, 2014. `http://stackoverflow.com/a/21212003/977026`.