CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Semantic rule engine for QML

Verifying semantic rules using computational tree logic

Master's thesis in Computer Science – Algorithms, Languages and Logic

Viktor Sjölind, Anders Johansson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Semantic rule engine for QML

Verifying semantic rules using computational tree logic

Viktor Sjölind, Anders Johansson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Semantic rule engine for QML
Verifying semantic rules using computational tree logic
Viktor Sjölind, Anders Johansson

Supervisor: Johan Thelin, Pelagicore AB
Supervisor: Wolfgang Ahrendt, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg

Abstract

One of the most common tools used for verifying code quality in programming languages are linter programs. However, in many cases, specifically in new languages, there is no such tool. An example of such a language is QML.

In this thesis this tooling issue is addressed by designing a rule engine, together with rules defined in a dedicated language. The idea is to use the rule engine as a framework for creating lint tools for any programming language and evaluate this by creating a sample implementation of a lint tool for QML.

In order to make the rule engine agnostic to which programming language that is being verified, the rules described in the rule language will describe structural restrictions of the abstract syntax tree produced from the source code. Moreover, in order to make the rule language as expressive as possible it is constructed as a dialect of Computational Tree Logic (CTL). Although CTL is a logic traditionally used to describe temporal properties of infinite trees, CTL fundamentally only describes structural properties of a tree and all its possible sub-trees which makes it a great foundation for our purposes.

An additional feature of using a dedicated language to define rules is that the rule engine becomes agnostic to what rules that are defined as well. Meaning that if a user wants to change, add or delete rules only the rule file has to be changed and nothing needs to be recompiled.

By using this approach, this thesis solves both the lint tool issue for QML and creates a framework for defining lint tools for any other programming language as well.

Keywords: CTL, QML, lint, static, verification, rule, engine.

Acknowledgments

We would like to thank Wolfgang Ahrendt who supervised this project and gave us guidance. We would also like to thank Pelagicore AB, especially Johan Thelin, Jonathan Pålsson, Tobias Olausson, Jeremiah Foster and Joakim Gross, for hosting us and providing us with both equipment and technical support.

Viktor Sjölind, Anders Johansson, Gothenburg, 2016

Contents

1 Introduction

This thesis is performed in collaboration with Chalmers University of Technology and Pelagicore. The purpose of the thesis is to research and construct a rule engine that can verify semantic rules on QML code.

1.1 Background

QML[1] is a user interface markup language used to implement graphical user interfaces for devices such as mobile handsets, in-vehicle infotainment systems and other embedded devices. QML is also a part of the Qt Framework[2] which is a cross platform application framework mainly used for developing application software with graphical user interfaces, but can be found in other non-graphical applications. The syntax of QML describes objects with properties and function bindings. The properties are used to define basic as well as layout structure and the functions are used to make the layout dynamic. The popularity of QML is rising and the need for better tools has risen with it.

A lint[3] is tool for examining source code based on different rules or criteria. QMLLint[4] is a program used to check the QML code for errors. However, as of now, the current implementation only perform very basic syntax tests. The tool runs the QML parser and then the QML lexer and check that no errors arose in the process. This is fine for simple programs but in more complex cases additional functionality is desired. Some lint programs like hlint[5], a lint program for Haskell code, checks the code for quality properties such as examine "risky code" for example code snippet that is known to be prone to generate bugs. Further hlint can check how much the code branches with cases such as if statements, code duplicity and code standards as well as checking that the syntax is correct.

Pelagicore is a company that works with open source solutions for infotainment systems for the automotive industry. While constructing a platform that runs QML-based applications, Pelagicore have noticed the need for improving the tools for QML and linting in particular. Pelagicore have a set of coding convention rules that they want their developers to follow in order to keep the code clean and structured and to avoid common mistakes. However, they have no tool that is able to verify that the developers comply to the rules. Furthermore, even if a developer would recall breaking a code convention rule at some point there is no way to locate where that was.

1.1.1 The QML language

QML[1] is a declarative language used for designing graphical user interface applications. A user interface in QML is built as a tree of objects, where each object has a set of properties, functions and sub objects. The different parts of this tree are defined in various ".qml" files, where each QML file contains a root object and import statements. The import statements represent either QML modules or folders both containing files that are accessible from the current file. Besides the explicitly stating imports via import statements it is always possible to access QML files that reside in the same folder as the QML file that is referenced from.

Figure 1.1 and figure 1.2 are two QML files that are connected through inheritance. In the file Main.qml the Qt Quick is imported and a root object of type *Rectangle*, a standard type, is defined. In the root object there are two sub objects of type *Element*. As these objects is of type *Element*, which is defined since there exists an "Element.qml" file, they inherit all of the properties of the root object in "Element.qml". However, QML also allows for both redefining properties and adding definitions for new properties not defined in the original scope. This is the case in both of the *Element* objects in figure 1.1. Both objects add a definition of a new property, the x and $anchors.fill$ property, and first of the objects also redefines both the *width* and *color* property.

In the case of static verification this means that the total scope, that is the QML current file together with all imported QML files, must be extracted and verified as a whole in order to be able to cover all possible cases were properties can collide. For instance, the property called $anchors.fill$ handles both positioning and size of an object using only a reference to another object. This means that the second *Element* object in figure 1.1 will have both the *width* and $anchors.fill$ properties defined which both define the size of the object. Depending on the implementation of the QML interpreter the same code will result in different results. This is an easy error to make since the properties are defined in different files but a lint program should be able to detect these kinds of errors.

```
// Main.qml
import QtQuick 2.0

Rectangle {
    id: canvas
    width: 800
    height: 200
    color: "blue"
    Element {
        width: 300
        color: "red"
        x: 500
    }
    Element {
        anchors.fill: canvas
    }
}
```

Figure 1.1: Structure of QML file Main.qml

```
// Element.qml
import QtQuick 2.0

Rectangle {
    width: 200
    height: 300
    color: "yellow"
    MouseArea {
        anchors.fill: parent
        onClicked: {
            var C = "green";
            parent.color = C;
        }
    }
}
```

Figure 1.2: Structure of QML file Element.qml

1.2 Problem description

There exist no publicly available lint tool for QML. This means that it is difficult to verify quality of QML code. Examples of such quality properties are code conventions and language restrictions. Code projects often have specific settings or "rules" that are important for that specific project and needs to be enforced. There exist lint tools for other languages that have adaptable settings for what to validate but those "rules" are often not customizable. It would be beneficial if the proposed lint tool could handle user customizable rules.

1.3 Main goal

The main goal of the project is to research and design the foundation for a general semantic rule engine for checking properties in QML. Where part of the research is on how to address the limitations of modern linter programs. The rule engine should allow developers to have total control over what properties the rule engine enforces as well as how these properties are verified. Furthermore, the rule engine should be designed as abstract as possible, easing the effort needed in order to use it for other languages.

1.4 Method

The method for designing a Semantic Rule engine for QML can be divided into 4 different parts; information gathering, designing a rule language, language case study and implementation of the engine. The information gathering phase will consist of understanding the problem of validating QML. The following questions need to be answered; How does other linter programs operate? What are interesting features that existing linter have problem with validating? What research exists in the field? The answers to these questions will be used to compose a base case list of violations and properties that are interesting validate. This list in conjuction with a literature study will then be the foundation for understanding what needs to be covered by the rule specification.

A rule language will be designed with the information gathered from the previous phase. The rule language will be designed to be as agnostic as possible to the language the linter is checking. In the language case study a simple implementation of the linter will be implemented in C++ and Haskell. The implementations will be evaluated against each other and one solution will be selected as the language of choice for the final product. In the implementation of the engine phase a final product will be designed and implemented. The final product will be tested using a test suit for validation of the engine. The constructed linter will als be presented to the QT community for more feedback.

1.5 Limitations

There are many interesting topics in the field of static verification of code. Unfortunately all of these topics can not, and should not be, covered in the same thesis. This is both due to time restrictions but also because there should be clear focus for each thesis. This section describes the limitations of this thesis.

1.5.1 Code duplication

Code duplication is identical or near identical code sequences and is generally considered as unwanted in any code. Locating identical sequences of code is an easy task. However, locating only identical sequences is often not sufficient, one also has to account for context. Moreover, there might be sequences of code which differ syntactically but have the same meaning and/or result. These occurrences are also to be considered occurrences of code duplication.

Using the approach of this thesis, checks for code duplication could be expressed to some extent. However, it would be quite restricted. Brenda Baker designed an algorithm that allows for thorough checks of code duplication[6]. This algorithm could be used in order to create an extra verification layer in the rule engine that locates code duplication. It is valuable to identify code duplication, it is not the main focus of this thesis. As such, checks for code duplication will not be a part of this thesis due to time restrictions.

1.5.2 Aliasing

Aliasing occurs when there are more than one ways to reference and manipulate the same area in memory. That is, there are more than one alias for the same memory area. This occurs naturally in many languages and in order to understand the true meaning of a program, analyzing the aliases might be required. As such it is also a topic that is interesting for the purposes of linting. For instance, one might want to be able to verify that certain memory areas or properties are only accessed using a specific API or method. However, analyzing aliases is a too big topic for the scope of this thesis and will not be covered in this project.

1.5.3 Focus on core functionality

Implementing a graphical user interface for the QRule Engine or integrating the QRule Engine into existing editors like Qt Creator, the main tool used in QML development projects, would improve the usability. However, the main focus in the project is on designing language for defining syntactic rules as well as developing the core rule engine, including algorithms for that will evaluate if rules defined in the this rule language hold for QML code given to the engine as input. Anything that is outside of that scope will not be a part of this thesis.

1.5.4 Implementation language

In order to determine the most suitable language to implement the rule engine in, two primitive implementations, one in C++ and one in Haskell, will be implemented. The experience and outcome from the case study implementations will select which language that is best suited for the main rule engine implementation.

1.6 Challenges

During the development of the semantic rule engine the following challenges need to be addressed.

1.6.1 Choosing implementation language

Evaluate what language to implement the rule engine in. As the QMLLint project is an open source project implemented in C++, implementing the rule engine in C++ as well would be beneficial in terms of the prospects of future maintenance. However, other languages such as Haskell might be better suited to implement a rule engine.

1.6.2 Declarative vs imperative languages

In a declarative language the written code describe what computations to do rather than how to compute something. Reactive languages keep bindings to variables updated as opposed to imperative languages where the expression is only evaluated once per call. Example $v = a + b$: In imperative language v will be assigned to $a + b$ but then a and b can change value afterwards without any effect on v. In an reactive language v will always be $a + b$, so whenever the value of either a or b is changed v will be reevaluated. QML is a language that is both declarative and reactive meaning that it has properties similar to a combination of HTML/CSS and ECMAScript (also known as JavaScript). As there is no defacto standard of how to write a semantic rule engine for such a language an analysis of the subject will have to be done.

There are some resemblance in the code structure between QML and CSS but they do not describe the same thing. QML is the structure with dynamic functionality together with design roughly like CSS combined with HTML. However, that is not two totally different structures.

1.6.3 Output from the rule engine

The error messages could be extended into several levels where the user can select what kind of output the Lint will use. This would make it easier for new users to understand how to improve their code and at the same time keep the advanced users happy. The problem is in identifying a well structured method for using several different levels of messages.

1.6.4 QML is a new language

The biggest challenge with the task of creating a semantic engine for QML is that QML is a fairly new language, and that it derives from a non-formal background. This means that there are few best practices and that the language, in some corner cases, can be ambiguous and is defined by the current implementation rather than a specification. This poses a challenge, but also an opportunity. As there is no current linter program, beyond the syntactical level, there is a great need for such a tool. At the same time, there are no conventions for how to interact with the linter program, e.g. how to embed hints in the source code indicating to the linter program that a rule should be ignored in a specific section. This, combined with the experiences from Pelagicore, gives the opportunity to create a very useful tool while approaching these problems in a QML context for the very first time.

1.6.5 Examples of semantic rules

Examples of semantic rules that are interesting to be able to verify by the semantic rule engine:

(a) Unused element id found, example fig 1.3
(b) Both anchoring and absolution positioning of a single element, example fig 1.4
(c) Element id outside file referenced (code convention violation), example fig 1.5

The rules describe different categories of code violations. Rule (a) is about finding redundant code. Rule (b) state that in this scope a certain property can only be declared once. Rule (c) state that certain properties may not reference outside of scope.

```
import QtQuick 2.0
Item {
    id: foo // Warn here, foo is never referenced
}
```

Figure 1.3: Structure of QML file with unreferenced id

```
import QtQuick 2.0
Item {
    id: root
    Item {
        anchors.fill: parent // These three lines
        width: 50            // are in conflict
        x: parent.x+20       // ^^
    }
}
```

Figure 1.4: Structure of QML file mixing anchors and fixed positioning

```
// Main.qml
import QtQuick 2.0
Item {
    id: root
    Element {}
    Item {
        id: someElement // Works only
    }                   // because this exists
}

// Element.qml
import QtQuick 2.0
Item {
    anchors.fill: someElement // Warn here, no way
}                             // to know that this
                              // element exists
```

Figure 1.5: Structure of QML file with reference outside of file

1.7 Context

When working on projects with a large code base and many developers involved, it is important to stick to code conventions in order to keep the code maintainable. However, in most cases it is not enough to setup code conventions and tell the developers to follow them, they have to be enforced.

Today there is no way to enforce code conventions on a QML project as the QMLLint only checks for syntax errors. This problem can be solved by defining a query language for semantic rules and defining the code conventions in that language. The complete set of rules could then be checked by an interpreter against the QML syntax tree generated by the lexer in QMLLint. Another way of describing the same thing is: extending the existing implementation of QMLLint with a semantic rule engine and defining a query language to describe the semantic rules in.

CSSLint[7] is an example of a linter for a declarative language. Although CSS is not a reactive language, there is useful information to gain from it. The rules that CSSLint check are defined as modules that are loaded into the lint. However, the rules are located in the source code for the linter. There are commands for ignoring unwanted rules. However, if a user of CSSLint wishes to tweak or create new rules the user has to change this in the source code for the whole program. Defining the rules as input to the linter would be an upgrade in this regard.

One aspect in building a good rule engine is to provide the user with quality feedback from the reviewed code. The messages must be written so that they are simple to understand by the developer. Often the error messages in a compiler is written by the developers of the compiler for debugging the compiler it self. This leads to error messages that are hard to understand and of little benefit for everyone but compiler developers. Javier Traver[8] suggest that it is possible to use several different levels of messages in the same compiler. The first level could be addressed towards the novice users with a certain set of messages while other levels could be addressed to more advanced users with messages containing information toggled towards their level. This approach would make it possible for the user to select what kind of information the lint will display and thus making it easier for new users while still addressing the demands of the professional developers.

1.8 Related work

As mentioned in section 1.6.4 the only linter program for QML is QMLLint[4] which only verifies the code on a syntactical level. Therefore, the only reference are linter programs for languages similar to QML. Since QML uses ECMAScript[9] for binding values and is structurally similar to ECMAScript code, linter programs for ECMAScript are interesting to examine.

1.8.1 ESLint

ESLint[10] is a linter for static analysis of ECMAScript. ESLint is designed to be highly customizable and is easy to extend with new project specific rules. Each rule in ESLint can be individually tagged to produce a warning or error. ESLint support a rule library by letting the user select which rules to include in each run by turning each rule on or off.

ESLint provides the ability to add and customize rules. It is similar to what this thesis tries to achieve. However, ESLint only solves linting in ECMAScript and as such the framework for defining rules uses ECMAScript specific constructs. This thesis aims to achieve a framework that allows for customizing rules without using language specific constructs resulting in a rule language that is agnostic to the programming language that is verified.

1.9 Connections to other logics

Computational Tree Logic (CTL) is used as base for the language for describing rules in this thesis . However, there are other logics that also might be suitable for designing the rule language. In this section such relevant logics are discussed and compared to CTL.

1.9.1 Ambient Logic

Ambient logic[11] is a concurrent calculus used for modeling computational modality of a system. The modality is concerned with between who and where a computation occur. Each computation is bound by its surroundings, called an *ambient*. An *ambient* is a place in which a computation can occur and the boundary of the *ambient* decide what is inside the scope. Some examples of *ambients* are; a web page (bound by a file), a wallet (bound by the person carrying it) or a train (bounded by the train tracks). Non-examples are a fire (can not be bound to a place) or logically related objects. *Ambients* can be nested together within other *ambients* and also be moved as a whole. For example a file (inside a laptop at work) can be accessed and edited at a new location (for example at home). Each *ambient* has a name, a collection of agents and a collection of *subambients*. An *agent* (or process) is a computation within an *ambient* that can affect the *ambient*, an example of a computation is to give the *ambient* the instruction to move. *Subambients* are a special type of *ambient* with a nested structure. They consist of a unique name, agents and further *subambients*.

Both Computational Tree Logic and Ambient Logic describe temporal properties of states and both support arbitrary steps in time. But Computational Tree Logic formulas targets states in trees where Ambient Logic formulas describe how computations effect *ambients*. As such Computational Tree Logic is more suitable to use when evaluating properties of syntax trees.

1.9.2 Spatial Logic for Trees

Spatial logic[12] is a logic for validating formulas on finite trees. In spatial logic, the truth of the formula depends on its location in the tree, meaning that the truth of a statement may depend on what nodes are in the tree above or below. In this sense Spatial Logic is comparable with Computational Tree Logic which talks about nodes along paths. A difference is that Spatial Logic is able to validate the placement of nodes. This is possible to some extent in Computational Tree Logic, however it is much more focused on how the structure expands in width and height. The extended support for validating placement that Spatial Logic offers could be useful to investigate ordering in code. However, the ability to express properties of how the trees are allowed to expand offered by Computational Tree Logic are more valuable in the case of linting programming languages.

1.9.3 P-logic

P-logic is a verification logic for Haskell[13]. Haskell uses lazy evaluation, meaning that properties or values in Haskell are not evaluated until needed. This means that a logic evaluation of a property p in Haskell can be either $True$, $False$ or $Maybe$, where the term $Maybe$ state that p is not evaluated yet and since can have any state. Given that P-logic is so closely coupled with Haskell it can not be used as a base for a generic rule language. However, if the verification algorithm defined in this thesis were implemented in Haskell, which it is not, P-logic could be used to verify it.

2
Implementation language case study

It is not a trivial task to chose what programming language to use for implementing a solution to a problem. In this project it is important to easily be able to parse abstract syntax trees (ASTs). Pattern matching is a design pattern that allows the developer to filter functionality based on the data types of the data that the developer wishes to parse. This is something that is very useful when parsing syntax trees. As a result pattern matching is regarded as one of the best way to approach language parsing. However, C++ is the main programming language used by the Qt community and as such also the language used to develop the QML parser. Since C++ does not natively support pattern matching it is not the obvious choice as implementation language for QRule Engine. As such, a case study have to be conducted in order to determine what language to use for implementing QRule Engine.

2.1 Language study

Haskell is designed to use pattern matching for evaluating what instructions to compute. Fig 2.1 demonstrates an example of a simple integer addition implemented in Haskell using pattern matching:

```
data Expr = EAdd Expr Expr | EInt Int

eval :: Expr -> Int
eval (EAdd e1 e2) = (eval e1) + (eval e2)
eval (EInt i)     = i
```

Figure 2.1: Integer addition in Haskell

If the input is constructed with the constructor EAdd then two contained expressions e1 and e2 will be evaluated and then added together. If the input is constructed with the constructor EInt the contained integer is returned directly. By using pattern matching, the code becomes easy to follow and easy to extend. For instance if the Expr language was to be extended with multiplication this can easily be done as follows:

```
data Expr = EAdd Expr Expr | EInt Int | EMul Expr Expr

eval :: Expr -> Int
eval (EAdd e1 e2) = (eval e1) + (eval e2)
eval (EInt i)     = i
eval (EMul e1 e2) = (eval e1) * (eval e2)
```

Figure 2.2: An extension of figure 2.1

Fig 2.2 is an example of how easy extending a function that parses expressions can be, assuming the function is defined using patternmatching. This example extends the function defined in figure 2.1 and introduces support for multiplication to the language.

Since C++ does not support pattern matching this is solved by use of the visitor design pattern illustrated by figure 2.3. The pattern is based on two key functions the *accept* and *visit* method[14]. The pattern is used for adding new virtual functions or operations to a class without modifying the original class. The visitor class implements all the virtual functions that are needed by the implementation. The Model in figure 2.3 has an *accept* method that takes a visitor as argument. The *accept* method then calls a *visit* method of the Visitor and passes the Model as an argument. This means that all functionality that can be visited in the Model needs to be covered by a corresponding method in the Visitor. By using the Visitor instead of the normal Model it is possible to add functionality to Model without altering the Model.

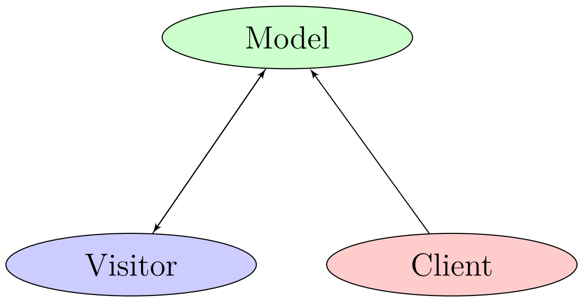

Figure 2.3: This figure describes the visitor pattern. The client calls $model.accept(visitor)$ then the model calls $visitor.visit(model\_data)$. Then the visitor parses the top structure of the model and calls any substructures recursivly by $model\_data.substructure.accept(this)$

2.2 Hypothesis

In the language case study there exist two main claims that need confirmation before the project can continue. Both claims significantly affect the layout and readability of the project. The hypothesis is:

- Since pattern matching is a really valuable tool when parsing any structure, good native support for pattern matching should ease the process of developing the rule engine. As such implementing the rule engine using Haskell should ease the development process compared to C++
- The Haskell implementation will be more readable and as such easier to understand than the C++ implementation.

In other words, Haskell will provide a better foundation for implementing this kind of software than C++ will. However, it is hard to tell how significant the benefit of having a complete QML parser from the start will be, which C++ does have.

2.3 Results

Both C++ and Haskell provide a reasonable platform for developing the rule engine. The benefit of using C++ is that the standard parser for QML is written in C++ and since the Qt framework is open source there is no problem in reusing it. The standard QML interpreter in the Qt framework uses a visitor pattern to parse the syntax tree. Visitor pattern[14] is a design pattern that makes it possible to do pattern matching in object oriented programming languages in a modular manner.

The parser and lexer for QRule is generated by BNFC[15] from the BNF definition of the QRule language. By using this approach the development process gets a head start. However, since BNFC generates all visitor calls with void as return type some manual editing is needed in the generated files. This extends the time required to process the consequences of changing the BNF definition. Compared to the Haskell approach this solution produces code with poor readability mainly due to very large files.

The native support for pattern matching in Haskell make the flow of the code much clearer and as such greatly improves readability compared to the C++ implementation using the visitor pattern[16]. One problem with implementing the engine in Haskell is that it is not possible to use the standard QML parser provided via the Qt framework, to solve this problem a separate parser for QML was implemented using BNF and BNFC. Implementing a new parser for QML introduce new complexity to the project. Since with this approach the new QML parser must be maintained and updated if the QML language specification change.

C++ was chosen as the implementation language for the QRule engine. The head start given by not having to implement and update a parser for QML and the fact that C++ is the language used by the Qt community are enough to compensate for the lack of proper pattern matching in C++.

2.4 Discussion

The functional properties of Haskell have native support for many of the aspects of doing computations on the structure of an AST. An AST is a structure with a limited number of components that need to be identified for the program to evaluate what to do next. Haskell has native support for this type of evaluation comparison to C++ where this has to be implemented by the programmer. The Haskell program therefore becomes much easier to maintain and understand, since the size of an average Haskell program only is about 1/10 the size of a comparing C++ program[16]. But C++ is the language of choice by the Qt community and since one of the purposes of QRule is to build an open source product which is maintained by the community, C++ was chosen as the implementation language for the QRule engine. However, as Haskell proved to provide a more readable solution it would be the implementation language of choice if the community did not have had an established preferred language. Though, by using C++ instead of Haskell there is no need to implement and provide a QML parser since it is possible to use the one in C++ provided by QT. This also means that if there are changes in QML in the future only the engine need to be updated since the parser is maintained by QT.

3
Rule language design

Traditionally linter programs use hardcoded rules to verify code. These rules can usually be toggled on and off but no further customization is possible. There are linter programs, such as ESLint[10], that allow users to write custom rules. However, while researching this field before starting this thesis, no linter program that offer a dedicated language for this purpose could be found. Such a rule language would be useful while creating a generic framework for lint tools. The focus of this thesis is to define a rule language dedicated to generic verification of programming languages.

In order to make a verification tool with a rule language that are truly agnostic as to what programming language that is being verified, there can not be any part of the rule language that describes any programming language specific constructs. Instead, the rule language should only describe structural restrictions that can be enforced on the abstract syntax tree.

This chapter describes how this can be designed as a dialect of computational tree logic with atomic expressions that access and describe properties of the nodes in the tree and first order logic.

3.1 Temporal logic

Temporal logic is used to reason about properties of a state transition system. In a state transition system a logic formula it self is not true or false as it is in propositional logic. Instead the formula is evaluated to be true or false in each state of the system[17].

3.1.1 Computational Tree Logic

One version of a temporal logic is computational tree logic (CTL). CTL is a logic describing temporal properties of tree structures. This can be used to verify *liveness* and $safety$ properties, that is, verify that certain patterns never occur and that some patterns always occur. A $safety$ property require that "something bad will never happen" and a *liveness* property require that "something will eventually happen".

All CTL expressions must consist of pairs of the temporal operators shown in figure 3.1. Furthermore, each pair must in turn consist of one member from each type. For instance, $AG\phi$ is a valid CTL expression while $FX\phi$ is not. Further examples of how CTL expressions are evaluated can be found in figure 3.2 and figure 3.3 verify expressions on trees.

Quantifiers over paths:
- Aϕ: ϕ holds for All subsequent paths.
- Eϕ: There Exists some path for which ϕ holds.

Quantifiers concerning specific paths
- Gϕ: ϕ has to hold Globally for all nodes in the path.
- Xϕ: ϕ has to hold for neXt node.
- Fϕ: ϕ has to hold for some Future node.
- ϕUφ: ϕ has to hold for all nodes Until φ holds.

Full grammar of Computational Tree Logic:

$$\{\top, \bot, \neg\phi, \phi\vee\varphi, \phi\wedge\varphi, \phi \to \varphi, AG\phi, AX\phi, AF\phi, A(\phi U\varphi), EG\phi, EX\phi, EF\phi, E(\phi U\varphi)\}$$

Figure 3.1: The full grammar of CTL together with an explanation of the temporal operators

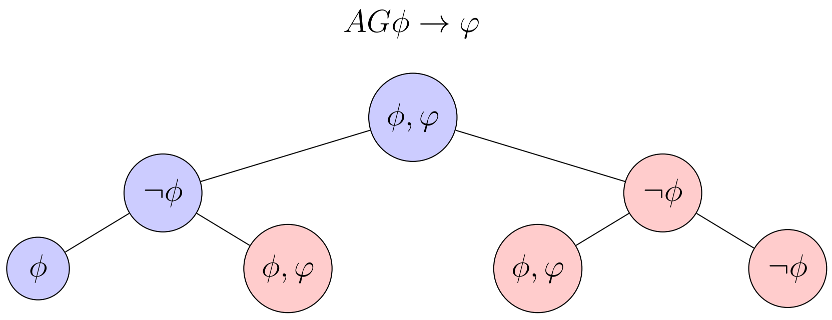

Figure 3.2: For all nodes in all paths, ϕ implies φ. In the example tree above, this is only true for sub trees, not the entire tree.

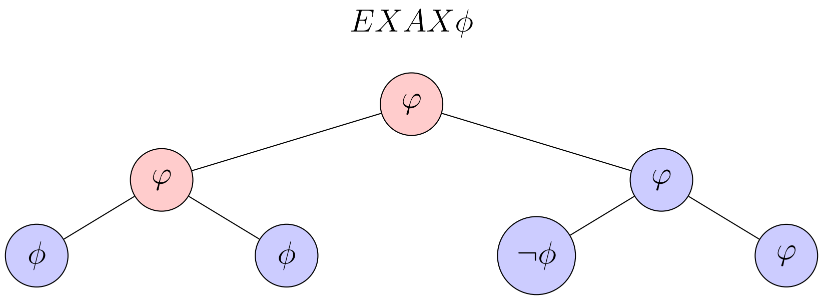

Figure 3.3: There exist a path such that from the next node on that path, ϕ holds for all next nodes. φ is used as a dont care term in this example.

3.1.2 QRule language

QRule is the name for the language used by the QRule rule engine, defined in chapter 4. The rules are designed to give flexibility and readability. A rule is defined by the following parameters.

$$Tag\ Severity\ RuleCause\ ASTScope\ Explanation\ \text{" :: "}\ Expr$$

Figure 3.4: The different parts of a QRule

The *Tag* is a unique string identifier for the rule, separating it from other rules in the XML output file. All errors found by the rule engine is sorted using the rule identifier. *Severity* is used to label and categorize rules into different categories, as either *Info*, *Warning* or a *Critical* rule. The different severity classifications are added to give the option for sorting rules into categories. The *Rulecause* can be either *language* or a *policy*. *RuleCause* is used to label the rule with a cause and to further separate the rules into categories. *ASTScope* can be either *File* or *Imported* and is used for specifying if the rule should be verified for a single QML file (*File*). *Imported* states that the rule will be checked against a single file and all files imported into that QML file. *Explanation* is a user defined text that explains the cause of the rule. Fig 3.5 demonstrate a example of a QRule for checking that all variabelDeclarations names start with a lower case letter. All parts of the rule except *Expression* is generic and the last part is a CTL expression formated for QML. *nodeType* is a member of the node interface and *VariabelDeclaration* is a node in the QML AST. The rule state that for each node in the QML AST of type *VariabelDeclaration* then the value of the node must match or start with a lower case letter.

```
"LowerCaseLetterStartVarName" Critical Policy Imported
  ?? "Variable names must start with a lowercase letter"
  :: AG nodeType = "VariableDeclaration" -> value match "[a-z].*"
```

Figure 3.5: Example of QRule for validating variabledeclaration names.

The rule language was designed with this structure since QRule needs to be adaptable and generic. By using a structure that is modular it is possible to add more keywords in the future for more functionality or better sorting of the output. It is also agnostic to what language the rules will check or validate. The *Expression* part of the language will be different depending on the layout of the tree but the engine algorithm and the QRule engine will remain the same.

3.2 Node interface

In order to generalize the verification algorithm used by the rule engine, an interface that describes how to access information from the nodes in the abstract syntax tree is defined. Using this interface, the verification algorithm can access information from the syntax tree without becoming dependent on the structure of the nodes in that particular syntax tree.

In the case of this thesis, an interface that is mapped to the atomic expression in the rule language is defined. This mapping makes it trivial to parse atomic expressions and access the required information from the node currently parsed.

The QRule rule language node interface is defined by the following different atomic expressions (the actual grammar is included as an appendix A).

- *value*, the value represented by the node accessed as a string.
- *valueType*, the type of the value represented by the node
- *nodeType*, the type of the node e.g. "VariableDeclaration" or "ObjectDefinition"
- *row*, row number at which the code that the node represents start
- *col*, column number at which the code that the node represents start
- *tokens*, tokens representing other language constructs such as keywords and symbols. Accessed by the $hasToken$ and $getToken$ atomic expressions in the rule language

Each node in a tree has a *value* represented as a string. Furthermore, each *value* has a type which is accessed by $valueType$. For every node there is also a type of the node it self accessed by *nodeType*. As an example; a node with type "VariableDeclaration" the *value* would be the name of the variable being declared and the *valueType* is "String". Furthermore, there are the two atomic expressions *row* and *col* that are used to access information about at which row and column the code represented by the currently parsed subtree of the syntax tree starts in the source code. Tokens are used to get information about keywords and symbols in the source code. For instance ECMAScript does not enforce the need to use semicolons. Tokens could then be used to check if there are any lines that do not end with semicolon.

3.3 Key elements of the rule language

This section discusses different aspects that the QRule language has to be able to solve and how this can be done using computational tree logic together with first order logic and atomic expressions.

3.3.1 Identifying leaves

Since code written in any programming languages always is finite, there will always be leaves in the abstract syntax trees representing the code. As such, for any language that describes abstract syntax trees it is important to be able to detect and approach about leaves.

In QRule detecting if the currently traversed node is a leaf of the syntax tree is done by using the following expression.

$$\neg EXtrue$$

Since the abstract syntax tree is finite, the only time the expression $EXtrue$ would evaluate to $false$ for any next node is if there is no next node. That is, when there are no next node the inner expression of EX, in this case $true$, will not be evaluated since there are no node to evaluate the expression against. Instead, the $EX\phi$ will always evaluate to $false$ when there are no next node, no matter what ϕ is.

3.3.2 Filtering on type of node

It is important to be able to restrict structure for certain types of nodes. In QRule this is done by matching $nodeType$ with either a string literal or a regular expression.

$$AG\ nodeType = A \rightarrow B$$

$$AG\ nodeType\ match\ A \rightarrow B$$

For all nodes in all paths, whenever we encounter a node of type that equals/-matches A the expression B has to hold from that node onward.

Following are some examples showing how this type of expression can be used:

1. If A denotes a function block definition then it is possible to set the maximum number of allowed statements to N by defining B as "$numberChildren < N$".
2. If A denotes a variable declaration then, assuming that the implementation of the rule engine defines the value of a variable declaration node as the variable name, it is possible that the format of the variable name by defining B as "value match$C$" which forces the regular expression C on the variable name.

3.3.3 First order logic

In order to describe recurrence of values or structures first order logic is needed. QRule defines forAll and exist quantifiers as:

$$\text{forAll}\ Ident\ in\ [filter] : expr$$

$$\text{exist}\ Ident\ in\ [filter] : expr$$

Where filter is a non-empty list of filtering strings interspersed with the "." symbol. The filter is used to pattern match what nodes to quantify over. Each string in the filter list denotes a node type. As a result, only nodes matching the last filter string with preceded by nodes matching the previous, if present, filter strings. This is illustrated by the following example:

$$\text{forAll}\ x\ in\ "A"."B"."C" : expr$$

In this example only nodes of type C preceded by a node of type B preceded by a node of type A will be quantified over.

$$\text{exist}\ x\ in\ "C" : expr$$

In this example all nodes of type C will be quantified over.

3.3.3.1 Accessing properties

In order to access properties of the node that is being quantified over the specified $Ident$ is used. For instance, if the $Ident$ is specified as x then the *value* property is accessed with the expression *x.value* in the quantified expression.

$$\text{forAll } x \ in \ "A" : x.value \text{ match } "[a-z].*"$$

This example forces all nodes of type A to have values starting with a lowercase letter.

3.3.3.2 Nesting

Like all other expressions in QRule forAll and exist expressions can be nested and describe substructures. An example of this is $AGnodeType = "A" \rightarrow \text{forAll } x \ in \ "B" :$ *expr* which quantifies over nodes of type B in all possible sub trees, of the syntax tree, starting with a node of type A. It is important to note that the root for the expression *expr* is determined by were the $forAll$ expression was executed, meaning that *expr* will start from the root in all the sub trees that are being quantified over.

3.3.4 The Labeling Algorithm

The labeling algorithm[18] for CTL model-checking uses a model M and a CTL formula ϕ to output the set of all states S of M that holds for ϕ. To check if a state s_0 holds for M we use the labeling algorithm to produce the set of all states that hold for M. Then it is easy to check if the state s_0 is part of the set S.
The labeling algorithm for CTL performs this check by executing the following steps.

1. Given a formula ϕ rewrite ϕ only using $\perp$, $\neg$, $\wedge$, EU, EG and EX.
2. Then call the model check algorithm.
   (a) Label the states of M with the subformulas of ϕ that are satisfied at that specific state. Starting with the smallest subformals of ϕ and working outwards to the complete formula ϕ.
   (b) After the algorithm has labeled the states with the subformulas of ϕ including ϕ itself. The algorithm output the states that are labeled with full formula ϕ

3.3.4.1 State explosion

Generating all the possible states of a formula is a problem since that the number of possible states increase exponentially with the number of state variables. For the Labeling Algorithm this is a serious issue that limits the maximum size of system that the algorithm will be able to validate. Consider a system with n processes that each can have m different states. This system can then be in m^n different states[19]. However the verification algorithm proposed in section 3.4 do not suffer from this problem.

3.4 Verification algorithm

By using a rule language that expresses requirements on the abstract syntax tree, these can be verified using the Labeling Algorithm. Unfortunately the labeling algorithm finds all nodes where the formula holds which is not the information requested by a linting tool. The naive approach then is to negate the expressions and use the same algorithm. This results in all nodes were the original expression does not hold. However, it is still too much information. The only nodes that should be found are the actual sources of the errors. The nodes that fail as a result of cascade effects from other nodes should not be listed.

Another approach is to simplify the Labeling Algorithm into returning the first node that does exactly match the given CTL-expression. This ensures that the output will contain only one of the actual failing nodes. By then removing the returned node from the syntax tree, the same algorithm can be called again using the pruned syntax tree. This process is repeated until there are no more failing nodes or there are no more syntax tree left to evaluate, the result will be the set of only the actual failing nodes. This is the approach used in this thesis and is described in more detail in figure 3.6.

```
function $Verify$(Expr $\phi$, ASTNode $\varphi$):
  /*
   * $\phi$ is the expression to validate
   * and $\varphi$ is the root node in the syntax tree
   */
  begin
    if not SAT($\phi$, $\varphi$):
      Out := [ $\varphi.lastVisited()$ ]
      $\varphi'$ := removeSubTree($\varphi$, $\varphi.lastVisited()$)
      return [ O : Verify($\phi$, $\varphi'$) ]
    else
      return [ ]
  end
```

Figure 3.6: The algorithm used to verify rules

When evaluating an expression ϕ on an AST φ the function SAT is called. SAT will then pattern match the head of ϕ and call the appropriate function in order to evaluate the tail of ϕ. For instance, if $\phi = EX\alpha$ then SAT will call $SATEX(\alpha, \varphi)$. This function is described in more detail in figure 3.7.

Figure 3.8, figure 3.9 and figure 3.10 describes the functions handling the normalized CTL expressions $EG\phi$, $EX\phi$ and $E\phi U\varphi$. All three of them use the same lazy approach; only continue evaluating the expression if there are no contradictions or the answer is not trivially true.

The default SAT function works the same way as the labeling algorithm does, converting expressions to instances of the minimal CTL set displayed in figure 3.11.

```
function SAT(Expr $\phi$, ASTNode node):
  /* SAT main function */
  begin
    case
      $\phi$ is AF$\phi$     : return SAT($\neg$EG($\neg\phi$), node)
      $\phi$ is AG$\phi$     : return SAT($\neg$E(true U ($\neg\phi$)), node)
      $\phi$ is AX$\phi$     : return SAT($\neg$EX($\neg\phi$), node)
      $\phi$ is A[$\phi$U$\varphi$]: return SAT(($\neg$E($\neg\varphi$) U $\neg$($\phi\vee\varphi$) $\vee$ EG($\neg\varphi$)), node)
      $\phi$ is EF$\phi$     : return SAT(E(true U $\phi$), node)
      $\phi$ is EG$\phi$     : return SAT$_{EG}$($\phi$, node)
      $\phi$ is E[$\phi$U$\varphi$]: return SAT$_{EU}$($\phi$, $\varphi$, node)
      $\phi$ is EX$\phi$     : return SAT$_{EX}$($\phi$, node)
      $\phi$ is $\phi \rightarrow \varphi$: return SAT($\neg\phi \vee \varphi$, node)
      $\phi$ is true    : return true
      $\phi$ is $\neg\phi$      : return S - SAT($\phi$, node)
      $\phi$ is false   : return SAT($\neg$true, node)
      $\phi$ is $\phi$ & $\varphi$  : return SAT($\neg$($\neg\phi\vee\neg\varphi$), node)
      $\phi$ is $\phi \vee \varphi$  : return SAT($\phi$, node) $\bigcup$ SAT($\varphi$, node)
    end case
  end
```

Figure 3.7: Normalization step that recursively calls the next stage

$$\{p, \top, \neg\phi, \phi \vee \varphi, EG\phi, E(\phi U \varphi), EX\phi\}$$

Figure 3.11: Minimal CTL Set.

```
function $SAT_{EG}$(Expr $\phi$, ASTNode node):
  /* SAT implementation for EG $\phi$ expressions */
  local var X,Y
  begin
      X := SAT($\phi$, node);
      if not X then
          Y := false;
      else
          Y := false;
          for each ASTNode child in node.children():
              X := $SAT_{EG}$($\phi$, child);
              Y := Y or X;
      return Y
  end
```

Figure 3.8: Algorithm for handling $EG\phi$ expressions

```
function $SAT_{EX}$(Expr $\phi$, ASTNode node):
  /* SAT implementation for EX $\phi$ expressions */
  local var X,Y
  begin
      Y := false;
      for each ASTNode child in node.children():
          X := $SAT$($\phi$, child);
          Y := Y or X;
      return Y
  end
```

Figure 3.9: Algorithm for handling $EX\phi$ expressions

```
function $SAT_{EU}$(Expr $\phi$, Expr $\varphi$, ASTNode node):
  /* SAT implementation for E($\phi$ U $\varphi$) expressions */
  local var X,Y,Z
  begin
      X := $SAT$($\varphi$, node);
      if X then
          Z := true;
      else
          Z := false;
          Y := $SAT$($\phi$, node);
          if Y then
              for each ASTNode child in node.children():
                  X := $SAT_{EU}$($\phi$, $\varphi$, child);
                  Z := Z or X;
      return Z
  end
```

Figure 3.10: Algorithm for handling $E(\phi U \varphi)$ expressions

4
QRule Engine

This chapter discusses the design and implementation of the rule engine that verifies rules defined in the rule language described in chapter 3 against an Abstract Syntax Tree (AST) produced by a parser for the language that should be verified.

4.1 QRule Engine overview

The following schematics of the QRule Engine describes the workflow from input to output.

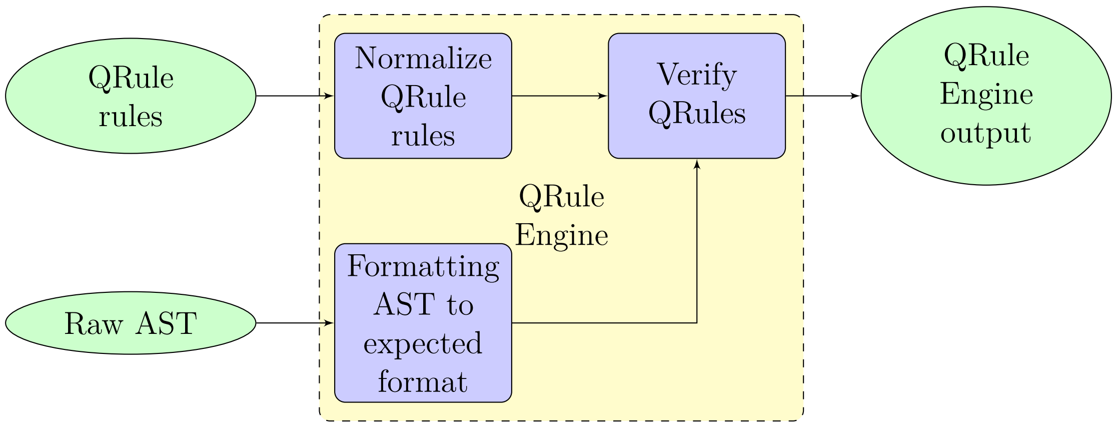

Figure 4.1: Overview of the QRule engine

The QRule Engine requires two input parameters; a raw AST produced by the parser for the target language to verify and a file containing the definitions of the QRule rule that shall be verified on the program. The raw program AST is wrapped into a format that is easier for the QRule Engine to parse. Then a super AST is constructed that includes all imported structures as well. Before verifying if the rules holds for the wrapped AST and the super AST, they are normalized to minimal form. During the verification process all rule violations are noted and after the verification process is done the noted violations are sorted by rule then added to the QRule Engine output.

4.2 Wrapping the syntax tree

Parsing the AST directly as it is produced by the parser for the language to verify, QML in this case, is tricky for several reasons. One reason is abstraction, that is the verification algorithm becomes dependent on an AST structure that is most often not maintained by the maintainers of the rule engine. Another reason is that writing rules toward a raw AST can get tricky since it may contain difficult structures such as linked lists. A third reason is that extracting information for each node visited becomes much more complex, which in turn worsens the readability of the code in the implementation of the rule engine making it harder to maintain the rule engine.

By introducing an interface for what and how information is be accessed from each node and sub-tree in the AST, the verification algorithm is abstracted from any language specific constructs. This interface also benefits from being closely coupled with the rule language. While formatting the raw AST received from the parser for the language being verified into matching the node interface, there is an opportunity to also remove redundant and overly complicated structures from the AST. By pruning the AST, the process of writing rules will become much more convenient.

4.2.1 Linked list

An example of an AST structure that, in the case of verifying with the rule language, benefits from being simplified is a linked list. As each node in a linked list both has a reference to the value of that node and the next node in the list, the depth of the structure depends on the length of the total length of the list, illustrated in figure 4.2. This property complicates the task of writing a rule that restricts how members of the list may be structured. By simplifying the linked list in the AST to a header node with all the members as children, as seen in figure 4.3, rule writing will become more intuitive.

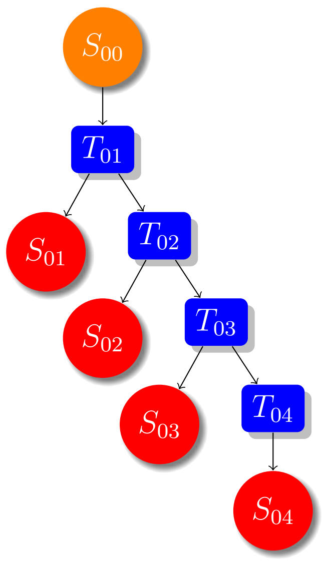

Figure 4.2: Example of a linkedlist in QML

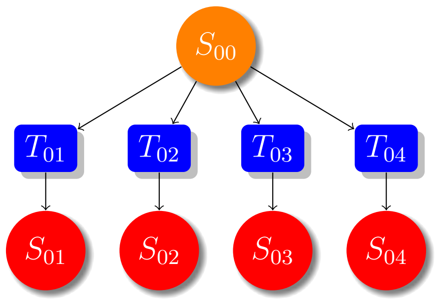

Figure 4.3: Example of a linkedlist after the wrapping, now prepared for CTL

4.2.2 Normalization

The QRules are normalized by the verification algorithm in section 3.4. The normalization process in the verification algorithm translate the CTL formula to the minimal set of CTL. Using normalization the algorithm only needs to cover the different states in fig 3.11.

4.3 Constructing super ASTs

As different rules are defined for different scopes, there is a need to be able to evaluate rules for a super AST. This is useful when there is a need for evaluating rules on the AST of the file currently being verified together with all the ASTs of the files imported from the current file and all their imports as well. This allows the rule engine to identify violations that occur when overloading, extending and/or overriding different structures.

In QML, structures are defined by objects containing properties, functions and other objects. The content of an imported object can be both overridden and extended in different ways at each point it is used. In order to be able to identify all instances where objects define colliding properties, it is crucial to be able to verify rules after the overriding and extending has been executed. An example of this can be seen in figure 4.4 where Main.qml imports two instances of Element that both inherit and override properties from Element.qml. One of the instances implements anchoring which collides with the x and with setup defined in Elements.qml. This is an error that is easy to make and creates a violation that lead to unreliable behavior where the property defined last will be the one used. However, by defining a rule that bans the presence of both *width* and $anchors.fill$ in the same object and verifying that rule in the *Imported* scope, this violation can be found.

```
// Main.qml
import QtQuick 2.0

Item {
  Element {
    x: 200
  }

  Element {
    color: "red"
    anchors.fill: parent
  }
}

// Element.qml
import QtQuick 2.0

Rectangle {
  x: 100
  width: 200
  color: "blue"
}
```

Figure 4.4: Main.qml has two instances of Element. Each instance inherits the properties defined in Element.qml but also override some property. The last instance also defines a value for a property not defined in Element.qml.

4.4 Evaluation order

The QRule Engine uses a depth first approach to verify the rules. By starting in the root of the first given AST and then recursively examine any sub trees, the engine is able to gradually generate and examine the context of a super AST. This is important in order to evaluate overridden and extended scopes. In the case of QML all the imported files have to be identified and evaluated before any other evaluation can be done.

The QRule engine QML implementation does this by using the following pattern:

1. Evaluate all imports not previously evaluated.
2. Parse the code into a raw AST.
3. Wrap the raw AST.
4. Construct a super AST by melding the imported wrapped ASTs into the AST wrapped in step 3.
5. Verify the rules on the wrapped AST and the super AST. Which one is used is determined by the AST scope for each rule.
6. Save the super AST constructed in step 4 with the filename as reference.

Using this pattern all the different scopes will be evaluated bottom up and each scope will only ever be evaluated once. This also guarantees that all possible sub trees of the complete super AST will be evaluated.

Due to the fact that different programming languages have different structures and meaning of importing and referencing files, the part handling finding imports have to be implemented separately for each programming language.

5 Conclusion

5.1 Results

5.1.1 A generic framework for creating lint tools

Due to the design of the rule engine and the wrapping of the raw Abstract Syntax Tree, the rule engine designed in this thesis can be used as a foundation for creating other lint tools. Lets assume that there is a need for a rule engine that can verify HTML code. As long as the new rule engine will be implemented in C++ as well, both the algorithm and the node interface can be reused as is. There are two parts of the rule engine that has to be reconstructed;

1. The part that handle wrapping and simplifying the HTML AST. Which produces an AST where all nodes match the node interface introduced in section 3.2.
2. The part that handle finding imported files and constructs the super AST.

Depending on how much simplification is done, the significance of the documentation changes. If there are only some slight differences between the structure of the original AST and the wrapped simplified only the differences might have to be documented, assuming there is a good documentation of the original AST. By further simplifying the AST the code will be come more intuitive for the end users while writing rules. However, as it is further away from the original AST it will become necessary to provide a standalone documentation, treating the wrapped AST as something entirely different than the original.

5.1.2 Unable to solve ordering

There are code conventions that states in what order different constructs should be defined. In the case of QML Pelagicore has rules for in what order functions and properties in objects are defined. There are no support for this kind of ordering enforcement in the rule language This is one of the drawbacks of using Computational Tree Logic as a base for the rule language since it is totally agnostic as to what order children to a tree is visited. However, although inconvenient, workarounds are possible while quantifying over all nodes in the scope where the ordering should be enforced and ensuring that nodes of specific types are defined before nodes of other types using the atomic expressions *row col*.

5.2 Discussion

This thesis has presented an approach to how a generic rule language can be designed and introduced algorithms for verifying that the rules defined in the rule language hold for any programming language. This section discuss the results presented in section 5.1 and the design choices made.

5.2.1 Reusing the rule engine

In 5.1.1 the steps to reuse the rule engine for linting another programming language were introduced. Following these steps requires some work, however it is far less work than writing a linting tool from scratch. The best way to approach this process is to start with making a trivial wrapper that does not simplify the structure. When the basic wrapper is finished, proceed with implementing the construction of the super AST. Using the basic wrapper and the ability to create super ASTs write rules in order to get a feeling for what would be a better structure of the AST. Then iterate this process, gradually improving the simplification. This is also the process used while developing and testing the QRule QML implementation.

5.2.2 Writing rules

While writing rules for the rule engine it is helpful to have a visual representation of an AST to look at. This need was discovered while implementing and testing the rule engine. As such support for generating representations of the super AST in the dot language was implemented. This has greatly improved the understanding of how the syntax trees work. If the concept of this rule engine is reused not using the implementation produced in thesis, providing the feature of generating a graphical representation of the AST is highly recommended.

5.2.3 Social and Ethical implications

There are multiple aspects of ethical implications that can be derived from the use of the QRule language and engine. While both the verification algorithm and the rule language are agnostic to what programming language, or other type of tree structure, is being evaluated, they are also agnostic as to what is the purpose of finding different kinds violations. As long as the data is structured as a tree the rule engine can verify properties of it, no matter the intentions or implications. However, this requires that the data is gathered in the first place. Since many data structures are in the form of, or can be translated into trees it possible to compare them and validate them using QRule. For example, if there exist a tree structure of emails it would be possible to enforce logical conclusions about the tree, that is what is written by whom and who knows what. Another example is using the rule engine to identify people with certain shopping behaviors.

5.2.4 Optimization

As the framework presented in this thesis is generic, some opportunities for optimizing the algorithms are missed. For instance, since a generic framework has to cover cases that do not exist in all languages a single purpose linting tool might not need to cover those cases. This applies to both the core engine and the rule language itself. As an example, if the rule language were implemented specifically and only for the scripting language Bash, there would be no need for the *valueType* construct as every value in Bash is a string [20].

On the other hand, with a generic linting framework any optimizations effect all linting tools implemented using that framework. Furthermore, if the same rule engine is used by many different linting tools, the incentive for optimizing the rule engine will be shared among many developers from different backgrounds. By releasing the rule engine using an open source license there is a great opportunity for creating a good community that will keep the rule engine up to date.

5.2.5 Readability vs expressiveness

By choosing Computational Tree Logic as base for the rule language, the rule language did inherit the great expressiveness from Computational Tree Logic. However, it also meant that the rule language did inherit the complexity of Computational Tree Logic as well. While presenting the content of this thesis both in academic and especially in industrial settings, one of the most common question received were; "Are there any plans for constructing a high level version of the rule language that is easier to use?". The answer to that question is that such a language were planed to be implemented during the course of this thesis but those plans were dropped due to time constraints.

The idea for the high level language were to use it as user interface that would compile down into the low level version that is more suitable for parsing and verifying. There were discussions about the syntax as well. Would one want to have a generic language that would work for all targets as the low level version does, or would a target language specific version be preferable. Making the high level language generic would be a really good way to improve the usability of the rule engine in general. However in order to make it generic it might not ease the process of writing rules as much as a target specific version would. For example a target specific version could have keywords and operators for handling common language specific constructs, such as traversing entire objects in QML, which would be cumbersome and require large setup otherwise.

This is still relevant for the future but requires more research in order to make the correct decision.

5.2.6 Choices done while designing the rule language

Although the goal always was to make a generic rule language, the initial approach was to use a more primitive language using constructs that were more reliant on the object structure of QML. This were quite effective in verifying basic properties of

QML but did not posses a lot of expressive power. As such there has to be a large number of atomic expressions with many of them overlapping.

In order to increase the expressiveness, it was decided to change approach and use a formal logic as base for the rule language. Both CTL (described section 3.1.1) and CTL* [17] were considered. CTL* is the conjunction of CTL and LTL (Linear Time Logic) [17] and allows to freely mix the different quantifiers without them having to come in pairs which is required in regular CTL. Using CTL* would give some extra expressiveness but would also add additional complexity to the rule writing process. Moreover, CTL* mainly provide a benefit while targeting infinite trees as much of the cases that regular CTL can not cover only appear in there. As such, CTL were chosen as the logic to base the rule language on.

5.2.7 Format of first order logic statements

De Bruijn index[21] is a method for removing variable or functions names and instead introduce numbers to refer to formal parameters. A reference to a name is bound to a λ and then referenced by "the distance" to the λ. By using "the distance" instead of a new name for referencing the λ, the name of the variable will always be fresh and no name capture is possible. For instance the function f. x .fx would be written λ01, where the 0 reference the innermost λ and the 1 the outer most. Instead of using De Bruijn index for naming the QRule $forall$ statements, the QRule language use "$for\ all\ x\ in\ y$" that is easier to read than λxλy.Since CTL is already hard enough for new users to understand we did not want to add more complexity to QRule. For QRule to be successful the QRules must be easy to construct.

5.3 Future work

5.3.1 High level rule language

As a result of constructing a very powerful rule language with focus on the being easy to parse rather than to be user friendly, it is not trivial to use for a rule developer. Therefore it would be helpful to have a high level rule language that compiles down to the rule language defined in this thesis.

However, it is not clear how to best approach the design of such a language. A language specific version could definitely make the rule development process more efficient. As an example; a QML specific version could define nested structures for objects and properties that represent how the QML code is allowed to be structured, in a way that is close to how QML code looks, making it more intuitive.

5.3.2 Ordering

The problem of solving ordering of code from an abstract syntax tree is not trivial. In a small AST with a node and two leafs, there exists no concept of ordering between the two leafs in computational tree logic. The wrapper used by QRule stores information about the line and row for each node. So it would be possible to compare all lines and solve the problem of ordering but this is not part of the current solution. Another problem is to limit the scope of what to compare and by this also limit the time complexity, since if the scope is too wide this will result in lot of unnecessary evaluations.

5.3.3 Multi level Output

The current version of the QRule engine only supports one text output from each QRule. The text output is defined by the *Explanation* part of a QRule (see Figure 3.4). This is a limitation in the model of the specification of the QRule, but can be solved by either altering the specification or attaching a separate output module. The QRule specification could be altered to take a list of explanations for each QRule and use a command line parameter to select the output level, and by output level toggle which text in the *Explanation* list to use. Another approach could be to separate the output and attach it as a separate module. In this case neither the QRule engine or the QRule needs to be altered but instead a separate module needs to be constructed. Both ideas for implementing support for multi level output have different pro and cons. The idea of altering the QRule and QRule language benefit from having one complete product that handles everything instead of adding separate modules for altering output. The altering solution would make a leaner product than adding another module but instead would make the rules larger and maybe not so easy to read and maintain.

5.3.4 Saving wrapped ASTs

The QRule engine does not save any data between executions of the program. It would be possible to add the functionality of saving each wrapped AST together with a hash of the file. Then QRule could first check if the file exist as a already wrapped AST. If the hash of the file is the same as the already saved AST the files are identical and no new wrapping is necessary. This approach could save time if the project is big enough so that there exist an large enough difference between calculate and compare hashes of a file than wrapping the AST of corresponding file.

Bibliography

[1] "Qml," http://doc.qt.io/qt-4.8/qdeclarativeintroduction.html, accessed: 2016-05-04.

[2] "Qt," www.qt.io, accessed: 2016-03-02.

[3] S. C. Johnson, "Lint, a c program checker," in *COMP. SCI. TECH. REP*, 1978, pp. 78–1273.

[4] S. M. (iamsergio), "Qmllint github project," http://code.qt.io/cgit/qt/qtdeclarative.git/tree/tools/qmllint, accessed: 2015-12-16.

[5] N. M. (ndmitchell), "Hlint github project," https://github.com/ndmitchell/hlint, accessed: 2015-11-27.

[6] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, ser. WCRE '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 86–. [Online]. Available: http://dl.acm.org/citation.cfm?id=832303.836911

[7] N. Schonning, "Csslint," https://github.com/CSSLint/csslint, accessed: 2015-12-16.

[8] J. V. Javier Traver, "On compiler Error Messages: What They Say and What They Mean." *Advances in Human-Computer Interaction*, vol. 2010, p. 26 pages, 2010.

[9] "Ecmascript," http://www.ecmascript.org/, accessed: 2016-06-01.

[10] N. C. Zakas, "Eslint," http://eslint.org, accessed: 2016-03-02.

[11] L. Cardelli and A. D. Gordon, "Ambient logic," *Mathematical Structures in Computer Science*, January 2006, to appear. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=67603

[12] C. C. Imperial, C. Calcagno, A. D. Gordon, and L. Cardelli, "Deciding validity in a spatial logic for trees," in *In ACM TLDI'02.* ACM Press, 2002, pp. 62–73.

[13] R. B. Kieburtz, "P-logic: property verification for haskell programs," 2002.

[14] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices.* Prentice-Hall, Inc, 2002.

[15] "Bnf converter," http://bnfc.digitalgrammars.com/, accessed: 2015-12-16.

[16] P. Hudak and M. Jones, "Haskell vs. ada vs. c++ vs. awk vs. ... an experiment in software prototyping productivity," Department of Computer Science, Yale University, New Haven, CT, Research Report YALEU/DCS/RR-1049, Oct 1994.

[17] M. Huth and M. Ryan, *Logic in Computer Science.* Cambridge University Press, 2013, vol. second edition.

[18] M. C. Browne, “Automatic verification of finite state machines using temporal logic,” Ph.D. dissertation, Carnegie-Mellon University, 1989.

[19] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, “Model checking and the state explosion problem,” *In 8th LASER Summer School on Software Engineering, sept 4-10*, pp. 1–30, 2011.

[20] “Ecmascript,” https://www.gnu.org/software/bash/, accessed: 2016-06-01.

[21] N. D. Bruijn, “Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem,” *Indag Math. 34*, pp. 381–392, 1972.

A
QRule language specification

This specification document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

The lexical structure of QRule

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a (ASCII) letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in QRule are the following:

`A`	`AF`	`AG`
`AX`	`Critical`	`E`
`EF`	`EG`	`EX`
`False`	`File`	`Imported`
`Info`	`Language`	`Policy`
`True`	`U`	`Warning`
`col`	`exist`	`existing`
`forAll`	`in`	`is`
`match`	`nodeType`	`nrChildren`
`possible`	`row`	`token`
`value`	`valueType`	

The symbols used in QRule are the following (formatted in ASCII):

`::`	`??`	
`.`	`(`	`)`
`+`	`!`	`=`
`<=`	`>=`	`<`
`>`	`&`	`|`
`:`	`->`	

Comments

Single-line comments begin with `#`.
There are no multiple-line comments in the grammar.

The syntactic structure of QRule

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*RuleSet*⟩ ::= ⟨*ListRule*⟩

⟨*Rule*⟩ ::= ⟨*Tag*⟩ ⟨*Severity*⟩ ⟨*RuleCause*⟩ ⟨*ASTScope*⟩ ⟨*Explanation*⟩ `::` ⟨*Expr*⟩

⟨*ASTScope*⟩ ::= `File`
 | `Imported`

⟨*RuleCause*⟩ ::= `Language`
 | `Policy`

⟨*Explanation*⟩ ::= `??` ⟨*String*⟩
 |

⟨*ListRule*⟩ ::= ϵ
 | ⟨*Rule*⟩ ⟨*ListRule*⟩

⟨*Tag*⟩ ::= ⟨*String*⟩

⟨*Severity*⟩ ::= `Info`
 | `Warning`
 | `Critical`

⟨*PathQuantifier*⟩	::=	`AG` ⟨*Expr*⟩
	|	`AF` ⟨*Expr*⟩
	|	`AX` ⟨*Expr*⟩
	|	`A` ⟨*Expr*⟩ `U` ⟨*Expr*⟩
	|	`EG` ⟨*Expr*⟩
	|	`EF` ⟨*Expr*⟩
	|	`EX` ⟨*Expr*⟩
	|	`E` ⟨*Expr*⟩ `U` ⟨*Expr*⟩
⟨*IAtom2*⟩	::=	`nrChildren`
	|	`row`
	|	`col`
	|	`(` ⟨*IAtom*⟩ `)`
⟨*IAtom1*⟩	::=	⟨*Ident*⟩ `.` ⟨*IAtom2*⟩
	|	⟨*Integer*⟩
	|	⟨*IAtom2*⟩
⟨*IAtom*⟩	::=	⟨*IAtom1*⟩
⟨*SAtom3*⟩	::=	`value`
	|	`valueType`
	|	`nodeType`
	|	`(` ⟨*SAtom*⟩ `)`
⟨*SAtom2*⟩	::=	⟨*Ident*⟩ `.` ⟨*SAtom3*⟩
	|	⟨*String*⟩
	|	⟨*SAtom3*⟩
⟨*SAtom1*⟩	::=	⟨*SAtom1*⟩ + ⟨*SAtom2*⟩
	|	⟨*SAtom2*⟩
⟨*SAtom*⟩	::=	⟨*SAtom1*⟩
⟨*Expr10*⟩	::=	`True`
	|	`False`
	|	`(` ⟨*Expr*⟩ `)`
	|	`(` ⟨*Expr*⟩ `)`
⟨*Expr9*⟩	::=	`!` ⟨*Expr10*⟩
	|	⟨*Expr10*⟩
⟨*Expr7*⟩	::=	⟨*Expr7*⟩ = ⟨*Expr8*⟩
	|	⟨*Expr8*⟩

$$\begin{array}{lll}
\langle \mathit{Expr6} \rangle & ::= & \langle \mathit{String} \rangle\ \texttt{is possible token} \\
 & | & \langle \mathit{String} \rangle\ \texttt{is existing token} \\
 & | & \langle \mathit{IAtom} \rangle <= \langle \mathit{IAtom} \rangle \\
 & | & \langle \mathit{IAtom} \rangle >= \langle \mathit{IAtom} \rangle \\
 & | & \langle \mathit{IAtom} \rangle < \langle \mathit{IAtom} \rangle \\
 & | & \langle \mathit{IAtom} \rangle > \langle \mathit{IAtom} \rangle \\
 & | & \langle \mathit{IAtom} \rangle = \langle \mathit{IAtom} \rangle \\
 & | & \langle \mathit{SAtom} \rangle\ \texttt{match}\ \langle \mathit{String} \rangle \\
 & | & \langle \mathit{SAtom} \rangle = \langle \mathit{SAtom} \rangle \\
 & | & \langle \mathit{Expr7} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Expr4} \rangle & ::= & \langle \mathit{Expr4} \rangle\ \texttt{\&}\ \langle \mathit{Expr5} \rangle \\
 & | & \langle \mathit{Expr4} \rangle \mid \langle \mathit{Expr5} \rangle \\
 & | & \langle \mathit{Expr5} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Expr3} \rangle & ::= & \texttt{forAll}\ \langle \mathit{Ident} \rangle\ \texttt{in}\ \langle \mathit{ListFilter} \rangle\ \texttt{:}\ \langle \mathit{Expr} \rangle \\
 & | & \texttt{exist}\ \langle \mathit{Ident} \rangle\ \texttt{in}\ \langle \mathit{ListFilter} \rangle\ \texttt{:}\ \langle \mathit{Expr} \rangle \\
 & | & \langle \mathit{PathQuantifier} \rangle \\
 & | & \langle \mathit{Expr4} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \mathit{Expr2} \rangle & ::= & \langle \mathit{Expr2} \rangle -> \langle \mathit{Expr3} \rangle \\
 & | & \langle \mathit{Expr3} \rangle
\end{array}$$

$$\langle \mathit{Expr} \rangle \quad ::= \quad \langle \mathit{Expr1} \rangle$$

$$\langle \mathit{Expr1} \rangle \quad ::= \quad \langle \mathit{Expr2} \rangle$$

$$\langle \mathit{Expr5} \rangle \quad ::= \quad \langle \mathit{Expr6} \rangle$$

$$\langle \mathit{Expr8} \rangle \quad ::= \quad \langle \mathit{Expr9} \rangle$$

$$\begin{array}{lll}
\langle \mathit{ListExpr} \rangle & ::= & \epsilon \\
 & | & \langle \mathit{Expr} \rangle\ \langle \mathit{ListExpr} \rangle
\end{array}$$

$$\langle \mathit{Filter} \rangle \quad ::= \quad \langle \mathit{String} \rangle$$

$$\begin{array}{lll}
\langle \mathit{ListFilter} \rangle & ::= & \epsilon \\
 & | & \langle \mathit{Filter} \rangle \\
 & | & \langle \mathit{Filter} \rangle\ \texttt{.}\ \langle \mathit{ListFilter} \rangle
\end{array}$$