The design and implementation of a Discrepancy Monitoring System

Managing the problems introduced by code duplication across languages

Kaz de Groot
3026027
k.degroot1@students.uu.nl
Utrecht University

First Supervisor: Jurriaan Hage
Second Supervisor: Slinger Jansen
Company Supervisor: Timo Bax

January 2, 2017

Abstract

With the reemergence of the client-server paradigm in the design of web applications, duplication of responsibilities is often inescapable, and we end up with applications reimplementing functionality in another language. In an attempt to lessen the impact of this sub-optimal design on an existing software product, multiple potential solutions were investigated and evaluated. Based on constraints posed by both the general case of an existing product and the specific environment of the case company, a tool using static analysis techniques was found to be the most promising approach. A proof-of-concept of this tool was developed, and tested against an existing application. While this tooling offered assistance and insight, the findings largely lacked the precision required to target specific problems, mainly pointing to problems with a broader scope, solvable with improvements in architecture and process.

Contents

1. Introduction

Modern web applications provide fully featured interfaces. To ensure the responsiveness expected of those applications it is sometimes necessary to duplicate some server side functionality in the user interface. As these interfaces have to be written in JavaScript, given the current state of browser technology, while server software lacks this restriction, the existing implementation often can not be used as-is. This necessary translation, no matter the manner by which it is achieved, creates an opportunity for problems.

As this ends up with multiple code segments implementing the same functionality, there is potential for discrepancies. The clearest case is one of the implementations being missing or broken. More insidious problems present themselves when both implementations function, but differ in ways that make the results inconsistent.

These type of problems were encountered at BusinessBase B.V. BusinessBase is a CRM consultancy, with about 30 employees, of which 4 are developers. This makes them the 2nd largest Microsoft Dynamics CRM consultancy in the Benelux. One competitive advantage is the use of in-house developed products to limit or even eliminate client-specific custom development. Partially by use of templates, but for the most part using their ExpressionEngine product. A type of Business Rule engine intended to allow consultants and end-users to implement business logic in a simplified, excel formula-like way, running both in the JavaScript based web UI for end users, and for applications interacting directly with the C# based server software. While the advantages for implementation are obvious, as could be expected, over time some discrepancies between the front-end and back-end results of certain rules were found.

Based on an initial review of professional and scientific literature, and interviews with relevant stakeholders at the case company, a tool using static analysis to detect potential sources of discrepancies was found to be the most promising, given the available options and the constraints of the company environment. A proof-of-concept version of this tool was developed and tested against the existing codebase. While this tool provided some interesting results and insights into the application, accurate determinations of equality were found to be out of reach for code with more than the slightest stylistic differences. Based on these results some initial proposals for improvement of code and process have been made, but as of yet these mostly lack practical effect.

Chapters 2, 3 and 4 will respectively define the actual problem, with some examples, show the context of the problem, both where it comes from and where we can look for a solution, and address some of the more philosophical consideration that need to be taken into account. Then both scientific and industry relevance will be addressed in chapter 5, followed by an outline of how the research was conducted in chapter 6. Then chapter 7 address the development process at the case company, how discrepancies might arise from current practices and the limitations it poses on solutions. Following this is chapter 8, showing how suitable the solutions described in chapter 3 would fit the constraints of an existing application and the specific requirements and constraints of the case company, and by extent, which solution is the most promising. Then chapters 9, 10 and 11 address the development of a proof-of-concept implementation of the most promising solution found. This encompasses some initial failings, lessons learned, a technical birds-eye-view of the final tool, and a more in depth look at how some of its tasks are fulfilled. To close it off, chapter 13 deals with the results. First some actual results of the tool with some additional context on their relevance and a discussion with some of the developers with regards to the tool results, other results of the research and their current and future impact on the development process. Then chapter 14 follows with a broader discussion of the results of the research. Chapter 15 address how the results answer the research questions in the conclusion, finishing with some potential avenues for future research in chapter 16.

2. Problem definition

As stated in Hunt & Thomas [10], within a program, every piece of knowledge should have a single, unambiguous, authoritative representation. More simply this idea is expressed as 'Don't repeat yourself' or 'DRY'. This is only partially possible when working with an application that incorporates the same functionality implemented in different languages, but the underlying concept has become even more important. The existence of a duplicate implementation neccesitates a single, unambiguous, authoritative representation of the functionality. However, because of the existence of multiple artifacts adhering to the specification, some mechanism should exist to monitor adherence to this specification, or rather discrepancies between the implementations and the specification.

While this monitoring could be conducted manually, and frequently is, this presents both a significant workload, and the nature of most of the problems is such that they might be easily overlooked. Especially when discrepancies are introduced during maintenance, when one method might be changed while the other versions remain the same.

The following two examples show how these problems might appear in practice.

Taxes

Because taxes deal in percentages, they can result in numbers with more than two decimals. For accounting purposes there are rules how they should be rounded (generally round up). But the IEEE Standard for rounding specifies the use of the 'Round to Even' method by default, while JavaScript just rounds up. Furthermore, depending on how the tax is calculated and some other details, either three or four decimals should be used, instead of the standard two.

Week Years

ISO 8601 specifies a number of properties of date notation. One of the more interesting concepts are the week numbers, and the related week year. Setup as a seperate date system from the standard year-month-date, it represents dates as year-week-date. Because the year only ends on the week end once every 7 years, we end up with a situation where the last few days of the year might be part of week 1 of the next year, or the first few days of the next year part of week 53 of the previous year. Because the week year is attached to the week, instead of the month/date, by incorrectly using the week year in a yyyy-mm-dd date format, those few days around new years would end up with incorrect years in their dates.

The objective of this research is to develop a tool that provides assistance in preventing, finding and resolving these discrepancies and implement it in an existing development process.

This is expressed in the following research question:

> How can we prevent introducing discrepancies in functionality between two similar implementations in distinct programming languages?

This can be further specified by the following subquestions

- What are the causes of discrepancies between multiple code segments?
- What techniques can be used to prevent or find discrepancies?
- How can prevention be addressed in the development process?
- Which types of discrepancies can be automatically detected?

3. Context

Problems as broad as the one addressed in this thesis do not generally have one clear solution. Furthermore we can be sure that, while no one true solution might exist, there will be stepping stones and building blocks providing insight into the shape a solution could take, the consequences of specific decisions, and novel approaches to specific aspects of the solution.

As such, this section will address research and industry knowledge in a number of fields presenting a road map to a solution.

3.1. Language requirements

The introduction of C and Unix between 1971 and 1973 heralded an age of portability in computing. If you could get an C compiler working for your platform, you could run Unix, and if you could run Unix you could compile (almost) any Unix program as long as you had a compiler for its language and your system. While this landscape has somewhat changed, with the addition of Windows as an operating system, and the move from Unix as a standard to POSIX, you're still pretty much set if your application interacts with the API of choice and had some sort of compiler or interpreter that addresses your computer's architecture. Over the past decade, however, we have seen the emergence of new platforms that have shifted these requirements. All the prevalent mobile platforms (Windows 8+, Android, and iOS) have a clearly defined preferred set of programming languages, with frameworks to match. Windows 8 and up require HTML5 for UI in their 'new' UI (that was known as Metro and Modern) and the application is expected to use the .NET framework and be written in a Common Language Runtime language. Android runs a variant of Java bytecode called Dalvik and has moved over time, from a VM to on-device compilation. Programs are expected to be developed against the Android SDK. iOS limits developers to the NextStep derived Cocoa framework, using either Objective-C or the recent Swift language. The web has a similar problem. While not closed by design, its standardization requires a higher level of abstraction, and though not necessarily planned, this has stuck us with HTML, CSS and JavaScript underlying everything that runs natively in the browser. Rather than an universal specification, there is a complicated web of support, features, interpreters, and innovations that is combined under the HTML5 moniker. Not really a standard, it is more a buffet of features, implemented at will by the multitude of browser vendors. And we should not forget that there is an entire subsection of devices that embrace these limitations by only supporting the web, like ChromeOS or, less successful, WebOS devices.

Not everyone is pleased with this state of affairs, and thus there is a lot of work aimed at side stepping these limitations. And so we find a number of solutions, either allowing deployment to multiple platforms from one source, by introducing some sort of intermediate language, or interpreter or just relying on the universal support of web technologies (eg. HTML5), or providing the ability to address the platforms API from a language of choice and the tool chain to transform that into something suitable for the platform of choice. These approaches work for mobile apps, however the solutions for web apps tend to be less complete. While some attempts have been made to allow authoring entire web applications in a language differing from the native languages, these approaches have not seen widespread adoption. Both Google Web Toolkit and Cappuccino still exist, but it is hard to find relevant examples of their use. Most web apps focus on building HTML, CSS and JavaScript individually, directing tool energy to improving the developer experience ('DX') of the specific language, rather than the entire platform. This has resulted in some trans-compiled languages for both HTML and CSS, that either add functionality, reduce verbosity, or both. Javascript, however, is where the real work is happening, with a multitude of tools and changes in environment that move JavaScript closer to being a type of assembly language, rather than a general purpose scripting language.

3.2. Self-imposed limitations

On the web a lot of the difficulties are self-imposed, because they offer great trade-offs. There is a balance to be found in everything, with the opportunity to shift as much or as little as wanted from the (mostly flexible) back-end to the (tightly controlled) frontend. Functionality on the front-end is more responsive, and off-loads your own infrastructure, at the price of limiting your development platform. But even that comes with its advantages, because the strict limitations are less the result of any oppressive corporate mandate, and more of de-facto standardization over a very wide range of platforms. This allows any well developed web application to run on almost any recent device running a modern browser. From computer, to mobile phone, to TV and back to an internet connected fridge or thermostat. And these web applications can compete with whats readily available in the native space. This is especially interesting for SaaS (Software as a Service) projects, where you can actually provide the entire application in the web-browser, avoiding complex setups for delivering native applications to your customers and keeping them current. And because everything is handled over the internet, together with the commoditization of other *aaS products, this creates low barriers of entry, both for providers and customers, largely eliminating problems with regards to scaling, geographical limits, environmental concerns, etc. Some popular examples are both the Office 365 and Google Apps suite, that offer purely webbased alternatives for traditional office applications, and for something more classical, Gmail, which since its introduction in 2004, has repeatably been found to outperform native alternatives.

3.3. Code generation

As mentioned, JavaScript has moved to a status as the de-facto (x86) assembly or VM of the web, which has created a wide market for tools that treat JavaScript like a compiler target. The further addition of node.js as a popular server side development environment has helped this, by providing additional incentive for a JavaScript centric workflow.

While a point could be made for investigating the other way around, compiling JavaScript to C#, the specific niche JavaScript has found for itself makes it a very ripe field. Beyond that, we can assume not every language feature offered by C# could be inferred during a (trans)compilation process, we would be hard pressed to use the full capabilities provided by C#.

Even though tools for compiling existing languages to JavaScript exist, the more popular approach consists of specifically designed languages that aim to provide a direct translation path to JavaScript, while enforcing some best practices or adding on features like a type system.

More interesting for our goals is the work that is being done in trans-compiling less specific, more widely used languages to JavaScript. Instead of designing a language specifically as a wrapper or superset, an existing language is used and translated into JavaScript.

First we will take a look at two new languages designed to target JavaScript. Then we will take a look at directly compiling C# to JavaScript through both a more generic tool, and a C# specific one.

3.3.1. CoffeeScript

CoffeeScript is a language inspired by Ruby, Python and Haskell, with a simplified syntax and additional syntactic sugar. As the language is trans-compiled, it is able to work around some of the more problematic aspects of JavaScript and can implement patterns that would normally be avoided due to complexity or verbosity. To provide some idea of the popularity of the language, it is currently the 22nd most popular language on github.

3.3.2. TypeScript

TypeScript was developed by Microsoft as a strict superset of JavaScript that adds functionality making it more suitable for use in large software projects. The most significant is the addition of a type system. Great care was taken in the development of the language to ensure interoperability with normal JavaScript code, including features that allow providing type information for existing JavaScript so it can be used without compromising the type system. We should not ignore, however, that not every JavaScript construct can be sufficiently described by TypeScript's type system. One advantage, especially for its longevity, is that most of the additions in TypeScript are based on future version of the ECMAScript specification, which implies that TypeScript code might be usable as native code in the future, without significant rewriting effort. It is currently the 26th most popular language on github.

3.3.3. Emscripten

Emscripten builds upon the LLVM compiler architecture, and takes the place of a compiler target. This in the same way as x86 machine code is a possible compiler target. Because LLVM is more like compiler infrastructure, rather than a classical compiler, it provides a halfway point in the form of an intermediate form (Intermediate Representation or IR) provided by one set of compilers, that can then be compiled to its target. Because of this, the same tool can be used for a number of languages, including, but not limited to C, C++, C#, Rust, and Ruby.[1] Because of this flexibility and the nature by which it is achieved, the code produced by Emscripten is styled much closer to the way one would write a low level C program or even assembly. The structure used being reasonably close to the way a computer operates internally, the specific patterns employed were formalized in the asm.js specification. asm.js is a very strict subset of JavaScript that enforces the format as currently produced by Emscripten, and by doing so allows any browser that implements support to employ a far more efficient runtime when working with asm.js compliant code. As a result, supporting browsers can run asm.js compiled C programs at as much as 75% of native speed, and this is still improving. Despite the advantages this offers, in the end, the code is still operating within the browser sandbox, and relies on Emscripten provided libraries to cover for the lack of native system libraries. Also, because of its nature as a compiler, we would have to deal with any dependencies on external libraries that might be involved. While a large part of the use of emscripten is as a proof of concept, compiling anything from mp3 libraries, to old games to work in the browser, there is some interesting commercial work using the tool. For example, Epic has used Emscripten to port a version of the Unreal engine to JavaScript, allowing developers to ship plugin-less, well performing 3D games to the browser.

3.3.4. Bridge.NET

Contrary to Emscripten, which is a more generic tool, Bridge.NET is a specific C# to JavaScript compiler. Based on the work done on Script# and Saltarelle, it aims to provide the ability to write JavaScript applications in C#. This is a decidedly different approach than Emscripten takes, and is more about developer experience than portability. Rather than producing asm.js compliant 'portable machine code', Bridge.NET aims for an as straight as can be translation, where one could mistake the end product as a (manual) JavaScript rewrite of the original C#.

Bridge.NET aims to provide JavaScript versions of large parts of the .NET library, based on Microsoft's Roslyn project, and offers out of the box interaction with JavaScript libraries like jQuery and Bootstrap.

While the two languages arguably have significant impact on code quality and developer experience, the fact they are generated from purpose built languages means this is all they do. Building a program in CoffeeScript or TypeScript might be easier than writing them in plain JavaScript, but they still are distinct languages, and thus require a distinct, additional code base.

Both tools for compiling C# to JavaScript are very interesting from a technical perspective, but would also require significant rearchitecting to be usable, in conceptually similar but practically very different ways. By requiring the exact same code to function in two entirely different operating environments an additional level of abstraction is required, to separate the i/o of the front-end and back-end runtimes, even if the functionality of classes is the same on a method by method basis. This isn't helped by the introduction of a custom compilation step into the process. Beyond adding complexity to the development stack, we introduce additional points of failure that are out of our control. And especially given their place in the process, these can create problems that are both harder to test for and and more complex to fix.

Though Microsoft's open-sourcing of large parts of the .NET standard library, and the related compiler architecture, has given a serious boost to projects that interact with .NET and C# code, most of the results currently look more experimental than practical. While it would be ignorant to dismiss the potential future impact of these projects, as it currently stands, they have not yet shown the improvements or maturity one would hope for. Possible consequences for future work will be addressed in chapter 14.

3.4. Code equivalence

If we still end up with multiple versions of a program, we should be looking to establish these multiple versions at least do the same thing. Establishing equivalence between programs is certainly not a new area of computer science, and a number of papers have been published on the subject. Most, if not all on the writing, however, has very specific requirements of the programs being compared. Use of functional programming languages is near universal, most likely as this allows for a more direct link to the mathematical semantics used to determine the equivalence. Certainly, most of the approaches presented lean far more towards mathematics than programming [19].

More interesting is the concept of bisimulation. It posits, simplified, that if you represent a program as a state transition system, if two programs consist of the same state transitions, what they do should be indistinguishable from each other. This approach, besides interesting, is also more suited to the semantics of imperative programming languages. Thus, if we could find some way to show that two different code segments were to be bisimilar, we could take it as given that whatever translation process, either automated or manual, had transpired, it worked as desired. However, while the concept applies, we find barely any mention in literature of practical implementations of bisimilation on actual applications. There are some papers describing the use of bisimulation on actual code, but those seemed to be more of a proof of concept, rather than a field ready framework that would be usable in a production environment [11].

3.5. Code verification

We see similar patterns if we move a step back to correctness or code verification. This is an area that is widely researched, as proving code to be correct is a very desirable capability. An example of the extent of practical applications of proven software can be found with Coq. More than just an automated theorem prover, it has the ability to generate a working application based on a proof. This moves beyond singular mathematical proofs; it has been used to develop a fully proven C compiler (CompCert) [12], and is used in a number of projects to prove the correctness of (subsets) of programming languages. It should however be noted that the programs can only be generated for languages like OCaml, and a proof needs to be provided. There has been some recent work in combining Coq in combination with CompCert to prove the correctness of algorithm implementations in imperative languages. For example, Appel [4] has proven the correctness of the SHA256 implementation used in the OpenSSL project. This, however, simultaneously shows how far we are still removed from using these methods in more practical environments, as this proof required a very specific tool chain targeted to C and a user capable of providing the proof.

So, at least currently, these approaches to verification and equivalence have very specific requirements of the tools used and the people using them.

3.6. Code checking

Moving on from more formal methods, we look to the tools that are more commonly used in development for providing some assurance of code quality. If we leave our desire for a **proof** of equivalence behind, and we focus on supporting development in order to reduce the impact that duplicating code might bring, some new avenues open up.

The most obvious example is testing, as it aims for the same goals as proofs would, though on a much lower level. Thus it provides a same basis for multiple implementations functioning similarly and detecting regressions. While a well maintained software project should already include comprehensive testing, there are some approaches that are of special interest to this case.

3.6.1. QuickCheck

QuickCheck is a tool developed by Claessen and Hughes [6], and is a very advanced test generation utility. In QuickCheck you define how the input and output of a function are related. It then uses random variables to verify that this relation holds, and if it finds an issue, it attempts to find the most simple input value that causes the problem. This approach was found to be very useful, as shown by the number of implementations there currently are, beyond the original Haskell version, and the commercial Erlang implementation. On the other hand, we find there doesn't seem to be an universal standard for how the tests should be defined, and while some versions seem to use similar approaches, none seem fully portable. There was some work done on a full implementation of QuickCheck in JavaScript, but this work has been rolled into a CoffeeScript-like language.

3.6.2. Test generation

Other approaches to test generation exist [7, 13, 16, 20] but generally present the same problems as QuickCheck does, while providing less functionality.

Furthermore, if we look specifically at unit tests, most tests should be relatively small, use a limited vocabulary and be largely identical (except programming language) across multiple implementations of the same functionality. There are existing techniques, both for generating tests in multiple languages and testing frameworks from a single representation in a testing DSL.

3.6.3. Program normalization

Even though current imperative languages can have lots of syntactic sugar and advanced language features, we can generally represent the functionality they implement by simplified pseudo code. Abelson & Sussman [21, ch 1.1] state that every language consists of

primitive expressions, which represent the simplest entities the language is concerned with,

means of combination, by which compound elements are built from simpler ones, and

means of abstraction, by which compound elements can be named and manipulated as units.

which implies that, while we might lose some niceties, we should be able to rewrite any program in any language to some simpler form that consists of some form of the three elements above, reducing it to something more like pseudo-code. Especially if we do not expect our resulting program to be in any form an actually compiling program, the simplification could and should very well result in a program that is far more suitable for comparison.

While there are existing tools for normalizing or simplifying code, we find that our goals for normalization eliminate all contestants found. Instead we focus on possibilities offered by working with more broadly usable, yet more focused tools for working with source code. While there are specific parsers for most languages, a tool like ANTLR can be used to generate an abstract syntax tree for just about any language, just like a compiler would, as long as it is provided the right grammar.

ANTLR

ANTLR (Another Tool for Language Recognition) is a parser generator, that takes a given grammar expressed in EBNF (Extended Backus Naur Form), and uses it to create a lexer and parser in one of its target languages. While previous versions have had wide language support, current versions only target Java and C#, with the further addendum that grammars can include ‘target-specific’ processing code that will be included in the generated code. This means that while grammars are available for a very large amount of languages, their construction will frequently limit the target language you can use. Though given that from incidental tests, most language specific grammars target Java as runtime language, which has limited impact on the portability of the resulting parser when compared to a C# targeted grammar. Furthermore, nothing prevents exporting the data structure produced by the parser and processing it with other tools, thus allowing flexibility in the platforms it is used for.

3.6.4. Static analysis

Static analysis is a single name for a number of techniques that work directly on the source code without requiring compilation. While in the most simple version this does not move beyond finding simple syntax errors and smaller examples of anti-patterns, other techniques can go much further. For examples, a tool like CodeQuest can be used to query the entire, annotated, source code of a program for very complex patterns [8]. Other techniques include a complete mathematical verification of all the functionality, along the lines described earlier. As we have discussed most of the mathematical aspects already, we will focus on the less formal approaches.

Linting

A class of tools named after their predecessor ‘lint’, a tool for flagging problematic code constructs in C language source code. Their use and functionality has expanded from a very focused goal to sets of tests that find problems with anything from anti-patterns, software metrics, style problems to counter intuitive language use; generally in a very extensible framework. To some extent, ‘linters’ currently cover most of the lower end of the static analysis spectrum. Their modular nature is very interesting, as it allows their use to be adjusted to the environment where they are used. As such, a linting tool might be employed by some just to address problems the original lint would have found, while others employ them as a one stop shop for all their static analysis needs. Current (optimizing) compilers frequently include tests and workarounds for a lot of errors a linter might detect, but this is less applicable to scripting languages, where compilation isn’t a factor, or development environments that lack a tight integration with the compiler which would prevent the ‘as-you-type’ feedback a separate linter might still be able to provide. Due to the nature of the problems they are generally designed to find, they tend to be very language specific, but their general design and architecture is very interesting for any tool dealing with some aspect of code quality.

3.6.5. Software metrics

Metrics are generally an easy way to judge something. If you can find a way to represent some type of information as a number, you can then add on some threshold values that are representative of some quality requirement. One frequently cited metric is KLOC (Kilo/thousands of Lines of Code) which can be seen as representative for the complexity of a piece of code, or for the productivity of a programmer if some time metric is added (eg. per hour or per month). Other, equally clear metrics are number of bugs, or even number of bugs per KLOC. These kinds of metrics are used, not only to judge the quality of code, or in some cases the programmer, but there are those, like the Jet Propulsion Laboratory, who set them as targets for their testing. Finding from statistical analysis over many years that there is a correlation between the quantity of code and the quantity of bugs, and not finding those bugs sooner implies a failing on the side of the testers, rather than impeccable programming. While this might promote the idea that everything can simply be reduced to a metric, reality is more complex. Brooks [5] famously states that even if programmer productivity could be fairly reduced to some measure of KLOC per month, that value is largely meaningless in a project context, and where reckless addition of development capacity will usually increase overall development time by introducing additional overhead. From this we see a similar realization, that might very well be informed by the Mythical Man Month, in the project management method Scrum, where, in most variants, the work allotted to a task in planning is represented as a point, an abstract representation of development effort that is primarily intended for prioritization, rather than as a shorthand for a specific number of man-hours. While over time some intuitive link between a point and actual time might emerge, that isn't representative. This is supported by the frequently espoused idea in the industry that an accurate estimate of a developers time can usually be arrived at by taking the estimate given and multiplying it by three. This is a long way to go establish two things: software metrics have their place, but it is very easy to ascribe them more importance than is deserved. Numbers can generally be used to explain just about any result you want. The other is that it is very difficult to quantify programmer productivity and code quality.

4. Defining equality

Showing the equality of things is a complicated problem. Not in the least because it is mostly a problem of scope and definition. This is where you quickly get to the difference between equal and equivalent. While equivalence tends to be a superclass of equality, the bounds of both are not very well defined and can be just about anywhere dependent on context.

To give some examples, we find that there is a point to be made for the relation between 'A' and 'a' being anything from equal, to equivalent, to not even that. Similar problems of classification would apply to '2 + 3' and '5', and '1.0' and '1', or less theoretical, two phones of the same type.

This invites comparison to Plato's concept of the 'idea' world that provides a metaphysical prototype of which real objects are instances, or more recently, type-token distinction, which presents a similar relationship. These have a rigid distinction between what fits into the one category and what in the other, but in doing so they instantly introduce an additional level of meaning and thus an aspect that can be compared. Even with a simple primitive like 'word', when comparing we now have to decide wether we judge equality on the type of type, a word, the type it is, 'word', or the specific instance, or token, 'word'. Depending on the choice, 'word' would respectively match every previous word, every previous 'word' or wouldn't match any thing at all, as it is an unique instance.

This problem directly emerges in computer science, where any sufficiently advanced programming language has to deal with the specific semantics of identity, comparison and equivalence it presents. This means that depending on the language, the built-in equality operator might work as expected on primitive types, but require the use of a helper method on more complex constructs, as equality there can be related to identity rather than value, which translates to comparing tokens rather than types.

By this we hope to show that finding equality, or even equivalence is less about fact, and more about defining what question you're hoping to answer. Taking the complexity involved even in comparing something as simple as a word or number, we can see that simply stating that two programs should be equal or equivalent might very well be too complex and ill-defined a question to answer. As such, we work towards some way to address the general problem in a more accessible way. While any question on the scope of an entire program, or even a distinct part introduces unmanageable complexity, due to the number of variables included, the first step already points us in the right direction.

For applications, we can take a functional approach to equality. As long as no side-effects are introduced, two functions are equal if for all input they both (deterministically) give the same output. Then if we take both functions to be equal, and we manage to find a counter example, that would point to a discrepancy. This, of course, is not a closed definition, as that would have any two functions that are not equivalent be discrepant. Thus we need some sort of definition. Based on problems that were encountered we find two cases. Either the discrepancy affects all results a little; for example as in a rounding error, where one function rounds towards zero and the other rounds from zero. Or it affects a few results substantially, like the difference between the year and the ISO 8601 week year, where for the days around January 1st they might be off by one.

5. Relevance

5.1. Scientific relevance

The current literature shows no practical ways of addressing the problems that were presented. While we can in no way deny that work has been done in this area, and techniques to solve problems similar to ours exist, they either have very specific requirements of the environment where they are implemented, or require resources that are unfeasible for most projects. We aim to present an approach that sacrifices theoretical purity to provide something that is more feasible in real world contexts. By prioritizing scalability and ease of use over quality and theoretical soundness, we can start off with a working product and then improve the quality of its results, rather than working on making a more theoretical approach function in practice.

5.2. Industry relevance

The requirements presented by platforms are becoming more of a constant which makes this problem ever more present. With some of the most important platforms currently available, limiting development to one specific language and framework or API, we are forced to sacrifice some freedom in the choice of our tools and languages. We have shown that there are a number of approaches available to mitigate these limitations, represented by multiple frameworks, patterns, and tools that attempt to solve the problem posed by code duplication across language barriers. However, as can be expected, these solutions are generally not drop-in. The use of a pattern needs to be taken into account in the architecture of the application, and thus needs to be decided on early in a project, otherwise they require large amounts of rework, if that is even feasible. The problems with frameworks are similar, in that they require a large technical investment to be implemented after the fact, if even possible. Beyond that, they generally have significant architectural requirements, that may make it very hard to find something that works in a situation that has specific architectural requirements of its own, like extending an existing application. This implies that both these approaches would probably involve rewriting large parts of the application from scratch, a decision that Spolsky [2] famously described as the worst strategic decision a software company can make. Tools have similar problems in the requirements department. They frequently are designed specifically for a chosen language and generally have some expectations of the code they apply to that do not exactly fit the current realities. By addressing this problem from the scope of an existing application, the tool and process are shaped by supporting an existing environment instead of carving their own. While this might not neccesarily result in a turn-key solution, it should provide a structured starting point for addressing these problems.

6. Research method

6.1. Case company

The practical side of this research was conducted at BusinessBase, a company providing CRM services based on the Microsoft Dynamics platform. During the development of their ‘ExpressionEngine’ module, which provides the opportunity to easily define new business logic for both client and server-side at the same time, some discrepancies between the results of the rules on the frontend and backend were discovered. Because the client-side operates in the browser, while the server-side is a component within a C# application, the business logic language needs to be implemented both in JavaScript and C#. Both the act of duplicating the functionality and differences in the languages themselves created opportunities for errors to be introduced.

6.2. Case selection

While this research aims to address a larger problem existing in the industry, it was triggered by problems encountered at BusinessBase. As such, the research primarily focuses on producing a result that fits the situation at this case company. Though some, if not most, of the results should be universally applicable, the proof of concept tools that are the primary result, would require moderate to significant rework to be effective unless in a situation that is unfeasibly similar to that of the case company. Also, given that the focus of this research is on the practical functioning of the tool and how that can function as a part of a more structured approach to preventing discrepancies, rather than the practical impact on the development process (as will be discussed with regard to verification) additional cases would have limited impact on the final product.

6.3. Validation

Removing validation problems in research with regards to development quality and productivity is generally hard, as active development has too many variables to control and establish a reasonable correlation between any potential changes resulting from this research. Problems with internal and external validity are largely avoided by working within well defined bounds.

6.4. Verification

We verify our results running the tool against the actual source code of the 'ExpressionEngine' and comparing the tools results for a random sample to a manual comparison. That should provide a more accurate image of real world performance, rather than correctly processing some pre made test cases.

6.5. Research method

The research follows the design science framework as presented by Hevner [9]. As such some tasks have had one or more adjustment iterations after their initial completion.

6.5.1. Identify problems

Both a literature study (chapter 3) and unstructured interviews (chapter 7) at the case company have been used to identify potential sources of discrepancies. The interviews were additionally used to clarify the impact of the multi language architecture and the possibility of discrepancies on both the application itself, product management and the development process. As part of the interviews, some initial problems with discrepancies were discovered.

6.5.2. Build proof of concept

Based on the results of the previous step and analysis of the source code a proof of concept tool was developed. The limited results from this version were discussed with the development team and used to inform both potential improvement in the development process and points of interest for the tool. As a result of changes building on this feedback, the proof of concept was largely scrapped (further discussed in chapter 9).

6.5.3. Build tool

Based on experience from building the proof of concept and better understanding of the underlying technologies afforded by an extended period of 'field research' at the case company, a second version of the tool was developed. Due to a more flexible architecture, this version of the tool is more suitable for iterative development.

6.5.4. Chart development process

Day to day experiences at the case company informed a number of issues in the current development process, that if addressed could help in preventing discrepancies being introduced during development.

7. Development process

While literature and personal experience are useful to set an initial scope, to find a suitable solution for issues in the development of an actual application by actual developers, it would be ignorant to exclude their experiences. Also, to gain a better sense of the real-world impact of the issues, opinions of 'end-users' could be useful. To that end, 8 unstructured interviews were conducted, each around 30 minutes. 6 with the developers at the company (even if they weren't primarily involved in ExpressionEngine development) and 2 with the consultants that had the most experience implementing the ExpressionEngine in client projects.

7.1. Developers

There is little deviation in opinions about the general development process. A new piece of functionality gets added to TFS as a 'Project Backlog Item', which should contain a fully defined functional specification by the time any developer would be able to pick it up. This combined with verbose naming conventions is intended to be sufficient to prevent the comment 'code-smell' [14], though no-one can readily explain how this covers either consistent rationale (why rather than what) or a consistent functional definition over time. There is a solid appetite for testing among all developers, though this halts at unit testing. While there is universal interest for implementing integration and regression tests, this is something that is still on the horizon.

When it comes to ExpressionEngine specific functionality, we quickly lose the two developers that weren't primarily involved in the development. While they are aware of potential problems, and even see the outline of a potential solution in better testing, they are, of course, less informed about the practical aspects of working on the specific code. The most interesting question is whether one of the implementations is leading, though the universal answer isn't very surprising. All developers agree that the C# versions is leading. Even going as far as noting that the functional specification is generally completed as part of the C# development. From there the stories start to differ. One developer mentions that development of both sides ideally happens concurrently, though the other interviews don't support this. Another developer adds that while C# is leading most of the time, sometimes the JavaScript is developed first and would then be considered leading. Checking this attack on the primacy of the C# implementation with other developers points us back to the PBI, rather than the specific implementation. Another slightly divisive issue is interplay between C# and JavaScript implementations. While everyone agrees there is no active effort to adjust the C# code based on decisions in the JavaScript implementation, the JavaScript developers feel their limitations are not taken into account when designing the C# implementation, especially when it comes to library functions, while one developer that primarily works on C# mentions eschewing library functions in order to better control the functions and prevent black-box issues.

When investigating what specific issues with discrepancies triggered the original interest in this kind of research, no specific example could be given. When drilling down, the primary cases of discrepancies that were encountered were found in problems with differences in default rounding algorithms and precision, and date/time issues, mostly related to using the correct timezone. Based on the developers, all these issues were caught before they could be released, so there has been little practical impact.

7.2. Consultants

The interviews with the consultants seem to confirm the conclusions from the developers that most issues are caught before they hit the users. While for both the number of projects they have used the ExpressionEngine on is still limited, neither could remember a case where their progress was hampered by a bug. This, however, did not mean there was no criticism. Both consultants were adamant over the lacking quality of the UI at this point in time. One interesting thing noted by one of the consultants was the speed by which new expressions were added. While, in part, intended to replace 'inline scripts', the lack of feature parity in some specific cases means sometimes manual scripting is still required. Especially given the lead time for getting a function from proposed to implemented and released tends to be long enough that the end of the project comes earlier.

7.3. Conclusions

Both the developer and consultant interviews show that time and/or resources are an important factor. Ability to implement is limited by the speed of development and implementation of better supporting (e.g. testing) techniques is limited by prioritisation of new features. This is understandable form a business perspective, as 'better tested' is hard to put on a factsheet. It also suggest that a solution that a time-intensive solution would probably lack the staying power required to be useful. While the interviews suggest that the current solutions mostly suffice in keeping problems out of the shipping product, we can find some issues with the process, however. A lot of the interaction between PBI and the two implementations assumes a theoretical purity that is hard to achieve in practical software development. The primary implementation concern of one of the implementations should be to function identically to the other one. Ideally that would match up completely with the functional description in the PBI, and based on the results it often seems to be close enough, but that should be a secondary concern. While in it self this is mostly a process issue, it does suggest that we look towards solutions that either justify this 'purity' or attempt to minify its impact by putting more focus on the interplay of both implementations.

8. Picking a solution

If we follow the definition of the problem to be solved as stated in chapter 2 we find that there is not a single problem to be solved, but there is more of a hierarchy of problems that offers the opportunity for solutions at multiple levels. Furthermore, as seen when discussing some of the approaches available, there is no silver bullet that offers a simple or comprehensive solution to the overall problems.

As we aim for a practical solution, in a somewhat specific environment, this also impacts the choices we make. The hope is that these limitations would make the solution more easily adopted in a real world context, contrary to a theoretically sound but complex to implement solution.

Given the options found, a selection needs to made to find the tools most suitable for use in this project, weighing the match to the requirements of the environment and the project. Furthermore, we need to take into account what further work a specific tool implies, both in modifying the tool to the work at hand and making the program suit the solution.

8.1. Protocol

Some striking similarities can be found between the solution attempted here and the construction of a software system out of available (commercial off-the-shelf) parts. While for the general structure of the research we stick with the Design Science method as proposed by Hevner [9], some of the more indept COTS selection methods address areas of interest that need to be investigated in forming a relevant set of criteria or requirements to judge possible component parts by. Furthermore, a generalized structure can be found describing the selection process that will be used to guide the selection of components and technologies that will comprise the final tool.

Navarette et al, 5 phases of COTS selection [17]:

- System architecture
- Requirements
- Market exploration
- Component evaluation
- Component selection

In this section we will address both system architecture and requirements. Market exploration was covered in section 3. Then we will move on to component evaluation and finally component selection. We note that in this case, components might both refer to an actual component (eg. software artifact, program) or to a more general technique or method.

8.2. Criteria

The criteria for selection were guided by the Inception phase objectives in EPIC [3] (emphasis ours)

> EPIC Inception phase objectives:
>
> - Establish a **common understanding of the scope of the solution** and its boundary conditions, including interfaces to systems outside the boundary of the solution.
> - **Outline viable candidate architectures**.
> - Differentiate the critical Use Cases of the solution, the primary scenarios of operation that will drive the major tradeoffs.
> - Identify and evaluate the **relevant segments of the commercial marketplace and other sources** for components and vendors/suppliers and negotiate tradeoffs among critical Use Cases, the candidate architectures, cost, schedule, and risk.
> - Exhibit and demonstrate at least one candidate solution against the critical Use Cases.
> - Estimate **cost, schedule and potential risks** for each candidate solution.
> - Determine **potential changes to the end user's business process** and **the tolerance for and inhibitors to implementing those changes** across the organization.
> - Establish a plan for acquiring any components and services needed for this solution.

Based on these objectives, we define the following questions we want to answer for every component we evaluate:

- How does this solve the problem?
- Does it fit the requirements of the environment?
- What is the impact on the development process? (Short and long term)
- What are the practical risks involved in adopting this solution?

We then assign a numerical value from 1 to 5 to each answer, based on how well it matches our expectations for the component, as composed based on the initial interviews with the developers at the case company and the authors experience in development.

While this means the valuation will be tailored to the context of the case company, we aim to explicitly note when a significant shift in valuation is derived from what was perceived as a case company specific reason, as compared to concerns and motivations that are more widely applicable to the development community.

That said, there is no true unified experience in the development space, and the options for a single digit sized development team are different from those of multi-billion enterprises (which is given shape by the fact that a number of the solutions presented here find either their origins or significant support in such organisations). While we expect size doesn't impact the reasoning itself, it would affect the underlying cost benefit analysis, with options easily discarded for a smaller company proving both feasible and interesting for a significantly larger one. (E.g. saving 1% development time accross the board, would merit a far larger investment at Facebook or Microsoft than at your local CRM consultancy.)

Another assumption made is that teams working on projects that have the problems described here, will, on the average, lack the expertise neccesary to build or design most of the tools and techniques mentioned below.

8.3. Basic approaches

Based on the hierarchy of problems we have found, we can subdivide potential components in those that address the more high level problem of having (to maintain) duplicate code and the specific problem we are trying to solve: preventing the problems that emerge from that duplication. Taking that division, we separate the components found into categories inline with those two descriptions.

- Components that help eliminating the problem
- Components that manage the solution

Here we take as a given that the problem we are trying to solve is more or less a symptom of the larger problem, so any true solution should address the larger problem itself and any solution solely addressing the symptom will always be a compromise.

8.3.1. Eliminating the problem

Beyond the code generation options described in section 3.3, we will address two more general architecture level solutions. While these are not backed by literature, we find that any discussion of the problem with domain experts generally elicits a mention of one or both of these solutions, so they should be addressed.

Unify language used on client- and serverside

If the problem is duplication caused by differences in programming language, one of the more obvious solutions is to stop using different languages. As the use of JavaScript is required for our use given the platform requirements of web applications, this would require to move towards a JavaScript based server application. As mentioned before this is a move that is currently very well supported by the larger JavaScript community, especially due to the emergence of the node.js platform. While some care would have to be taken to make the same library portable between the server and the client side, as long as modularity is taken into account during the design that should not pose any significant problem. We should however recognise this is not an easy task, and any design of this new, sharable library would require complete abstraction of input and output to be able to function equally well on front-end as back-end. This pattern is dubbed Isometric JavaScript, and has been implemented in various degrees in multiple frameworks and production applications, moving from some shared libraries to a hybrid stack where most processing can be done both server and client-side.

How it addresses the criteria	
How does this solve the problem? It eliminates the problem, because there is no duplication anymore.	**5**
Does it fit the requirements of the environment? It does not. The server side language choice is generally and certainly in this case made for good reasons. In this case the serverside language needs to be a CLR language, which, while not limited to C#, does exclude JavaScript.	**2**
What is the impact on the development process? (Short and long term) Even if it was feasible, it first would require a complete rewrite of the serverside of the application, and a partial rewrite of the client side. Furthermore, this requires the existing developers to be(come) fluent in the new language, which would require some adjustment time, or actually replacing developers, in order to limit the the effect on productivity.	**1**
What are the practical risks involved in adopting this solution? If you manage to get it working, it largely circumvents any problems normally presented by this scenario.	**4**

Avoid clientside processing

Given that the move towards rich UI web applications was at least partially motivated by the options provided by the XMLHttpRequest API introduced to the general public by Microsoft Internet Explorer 5, offloading some information processing to the server is certainly not unheard of. Furthermore, it is to be expected that within a sufficiently complex client side program, there will already be some parts that require a roundtrip to the server, for access to information not available on the current page. As such, this is a fine approach when applied to applications of limited complexity, where the the number of possible interactions is fairly limited, but is complex to scale. It still requires some part of client side technology that addresses both interaction with the UI and the external web service. This client side component doesn't have to be complex if the functionality required is limited, but once we are working with larger sets of data, more points of interactivity and even interconnectivity, we move from being able to treat specific interactions as functional and atomic, to having to worry about state, race conditions, and side-effects. When we mention that this technique is hard to scale, it's not just about the complexity of the program, but also about the user experience. This is fairly important, as user experience is one of the primary reasons web based user interfaces are widely accepted. Nielsen [18] states that response times of an action should be kept under 0.1 second or 100ms to be experienced as instant by the user. This means that if moving a processing step to the server side would push the total duration of the action over that 100ms barrier it severely impacts the user experience. If we then take that any action might fire a number of different actions that might not be parallelizable, we would need any response to be comfortably under the 100ms limit. Given that just internet latency will easily add 15-30 ms to each request, combined with a multi user scenario, this poses an interesting engineering challenge. Especially with complicated, geographically diverse scenarios, the quality of service required to make this work becomes quickly unfeasible.

How it addresses the criteria	
How does this solve the problem?	**4**
It largely eliminates the need for duplication, though depending on the level of abstraction achieved in the client-server interface, new functionality might require client side changes.	
Does it fit the requirements of the environment?	**3**
As the technique is mostly platform agnostic, that doesn't pose any problems. The performance problems presented could pose a problem, as they partially undercut the usability of the client side part.	
What is the impact on the development process? (Short and long term)	**3**
Ideally it would be a one time change, though new functionality might require changes in both sides of the client-server interface. We should not understate the size of this change, which would most likely require replacing most if not all of the client side code and creating a suitable endpoint on the server side.	
What are the practical risks involved in adopting this solution?	**3**
The main risk is decreased performance. Depending on a number of variables real-world performance might be significantly slower than test scenarios, when dealing with multiple concurrent users and more complex interactions.	

Generate the JavaScript from existing code

This section will only discuss Emscripten, as the other three code generation approaches presented in section 3.3 do not allow compiling existing program code to JavaScript, but either require a specialized language or special annotations in the C# code. As such, they will be discussed in section 8.3.2 instead.

Emscripten Emscripten is an interesting approach as it should allow the direct compilation of the existing C# code to a JavaScript module that would offer the same functionality client side as server side. The same caveat as with the other methods applies here too, both the client and serverside need to provide some common way of interacting with the rest of the application, so the specific module that is to be translated needs to be portable between both environments. A more significant disadvantage is a direct consequence of the compilation. While applications would generally include library functionality by including a version shipped with or available on the computer, this is impossible in JavaScript beyond the library offered by the JavaScript implementation in the browser. As such, any attempt to compile the library that you would like to migrate to JavaScript would require compiling any library it used. While this might be acceptable for more resource intensive programs like videogames, where this additional overhead would only be a drop in the bucket, it presents a significant performance impact for a lighter web application, both in retrieving the libraries and in executing them.

How it addresses the criteria	
How does this solve the problem?	**4**
Eliminates the duplication problem by producing the JavaScript version automatically based on the C# code	
Does it fit the requirements of the environment?	**4**
Yes. It would eliminate a lot of the JavaScript work, and once the process is in place should function unattended.	
What is the impact on the development process? (Short and long term)	**3**
The Javascript part of the application would need to be changed to be able to deal with an automatically recompiled version of the code, as the structure isn't the same. Care would need to be taken in developing the transcompiled parts of the application, that no features unsupported by the C# compiler used for Emscripten are used.	
What are the practical risks involved in adopting this solution?	**2**
The Emscripten tool becomes an essential part of development, and needs to be considered beyond reproach. If either the C# to IR or IR to JavaScript compiler introduces a bug, these will not only be hard to find, but also hard to solve as patching the JavaScript would have to be redone after every recompilation and solving the problem in the compilers is not feasible for developers without a deep LLVM background. Also, because all the executable code needs to be compiled and provided, this could limit the use of certain external libraries, beyond providing every user with a moderately obfuscated version of a large part of the application.	

8.3.2. Managing the problem

Moving on from the more significant, impactful solutions, we move on to the more supportive components. While none of these options will eliminate the need for the code duplications, they all present opportunities to mitigate the impact of the duplication on the code quality and the development process.

Generate the JavaScript from new code

We find all the methods presented in section 3.3 provide distinct advantages over using plain JavaScript, though in the case of Bridge.NET, we still require a solid understanding of the way JavaScript works and interacts with its environment for development. On the other hand, they, with the possible exception of Bridge.NET, do nothing to mitigate the code duplication issue itself. So interest for these tools comes mostly from an idea that while not neccesarily causing discrepancies, the language quality of JavaScript doesn't help.

Bridge.NET

Bridge.NET might prove to be the most interesting of these options, as it leverages both existing knowledge and the functionality of C# to improve developer experience. Furthermore, duplicating code would in this case really be duplicating, as for most cases it would be copying the existing code, modifying it to work in a browser context and adding the requisite metadata for the compiler.

How it addresses the criteria	
How does this solve the problem?	**4**
It removes the need to write the duplicated code in JavaScript, which means code can actually be copied, and ideally the language is both safer and more familiar, reducing the chance for discrepancies.	
Does it fit the requirements of the environment?	**4**
It is a .NET application with native Visual Studio support. It doesn't require significant developer experience.	
What is the impact on the development process? (Short and long term)	**2**
All the duplicated JavaScript that currently is part of the application would have to be rewritten as C#. To what extent this can be achieved by copy and pasting the existing C# implementation is hard to gauge in advance. As the Bridge.NET C# code still natively interfaces with JavaScript, a decent understanding of the libraries involved in that is necessary. Furthermore, to get most of the advantages of being a typed language, headers would need to be made for any external library that is used for which one is not provided already. Also, the C# that can be written is limited by the compiler's support, which is not universal. This means that some more modern features of the language need to be avoided to make the cross compilation work.	
What are the practical risks involved in adopting this solution?	**2**
As with emscripten, any bugs resulting from the compilation process would be hard to find and hard to fix without working around it.	

TypeScript

TypeScript is mostly interesting because of the broad support from Microsoft. While the extra functionality provides the opportunity to write better, safer code, which could very well resolve some of the sources of discrepancies, it isn't a panacea. Because it is a JavaScript superset, rewriting would not be required, but at the same time, without rewriting, it would not offer any advantages. In theory, the modifications would be limited to some minor refactoring to use the new language functionalities, but realistically it is hard to gauge how effective some minor changes would be without some more significant restructuring.

How it addresses the criteria	
How does this solve the problem?	**2**
It offers a nicer, more advanced language than JavaScript, that might catch some of the elementary discrepancies, and might allow for some simplification of existing code that deals with problems now taken care of by the language. Beyond that it does not offer any inherent advantages.	
Does it fit the requirements of the environment?	**5**
TypeScript is included in Visual Studio, so it wouldn't require external tooling, with the official Microsoft support being a big plus. This only applies in Microsoft centric environments.	
What is the impact on the development process? (Short and long term)	**4**
Some refactoring or rewriting will be required to see any advantages. Because of the inherent compatibility with JavaScript, this doesn't need to be a big bang change, but any advantages will be limited until the entire codebase is ported.	
What are the practical risks involved in adopting this solution?	**5**
The current implementation is supported by Microsoft and the changes are based on those proposed in a future version of the ECMAScript specification.	

CoffeeScript CoffeeScript is mentioned, because even beyond TypeScript it is by far the most used transcompiled JavaScript variant. It's mostly a thick syntactic frosting covering the language, intended to simplify some common usecases in modern development for the web (first-class functions, callback heavy design, functional programming inspired patterns). While this definetely makes the language easier to write and will prevent some common problems; it also is an entirely new language, that offers familiarity only to those experienced with languages like Ruby, Python, and Haskell, more than C.

How it addresses the criteria	
How does this solve the problem?	**1**
Beyond an arguably better syntax, and some shortcuts and safeguards for common anti-patterns, CoffeeScript offers little to no advantage in reducing discrepancies for a project like this.	
Does it fit the requirements of the environment?	**2**
CoffeeScript fits well into an environment where the other languages used match its syntax and semantics or its the only language. In most other contexts its use makes little sense.	
What is the impact on the development process? (Short and long term)	**2**
As it's a new language, it would require a complete rewrite of clientside code. Beyond that, it is significantly different in syntax from JavaScript, so it would require some amount of retraining to get familiar with the language, and possibly even more to actually take advantage of its features.	
What are the practical risks involved in adopting this solution?	**5**
CoffeeScript is used by a number of very large organisations that support the development. Furthermore, the JavaScript it produces is generally of similar quality to that of a decent developer, so fixing incidental problems or moving away from CoffeeScript is feasible.	

Code verification

While there are multiple examples presented in the literature, we will only address Coq. It came up most frequently in the initial exploration of the problem space, and while other tools and approaches exist, aiming to provide comparable functionality, a cursory analysis of their working suggests that none would significantly improve on this approaches pain points. These would provide an interesting avenue for further research. As described earlier, some interesting advances have been made with the use of Coq, especially in combination with CompCert. But as much as it supports that verification of real programs is possible, it also clearly points out its current limits. Setting aside Coq's ability to generate OCaml from a proof, the opportunity to prove C programs seems interesting. There are a number of asides, though. The proof itself, of course, is not trivial. And while the promise of C support is met, it does require a translation step from the standard C's that are used to a C subset known as verified C (which is the C supported by CompCert). Of cource, C is still a vastly different language from JavaScript and C#.

How it addresses the criteria	
How does this solve the problem?	**5**
If both implementations were to be verified by a proof, there would be no discrepancies.	
Does it fit the requirements of the environment?	**1**
In no way. Not only are both the languages used, with special note of JavaScript, not supported, this method would also require full access to the underlying frameworks to perform as expected, which is unfeasible.	
What is the impact on the development process? (Short and long term)	**1**
Setting aside the language requirements, which would be hard to impossible to overcome, constructing proofs would require both significant skill and time investment, not commensurate with the resulting advantages. Without significant training it is to be expected that no one on an average development team could provide these proofs, especially without significant impact on their other duties.	
What are the practical risks involved in adopting this solution?	**3**
This requires a very complicated, largely experimental technical stack to function, but doesn't affect the code itself. It doesn't introduce risk beyond the possiblity of it failing.	

Bisimulation

While the theory here seems suitable, we find that practical examples are lacking. Furthermore, it seems that with an application of a reasonable size, the transformation for bisimulation to work as it is intended, would be complicated and computationally complex. Beyond that, we fear that it would be hard to retain the metadata that would be able to trace back an erroneous state transition to its originating source code. Also, there is a reasonable fear that any differences mandated by the language would be amplified instead of ignored during the conversion process, which would trade false negatives for new false positives. Another problem that we find with most of these lower level solutions is that there is barely any way to effectively deal with the existence of external libraries. Especially with language features offered in most newer languages, such as reflection, just-in-time compilation, etc. a reliable transformation to some deterministic statemachine is unfeasible.

How it addresses the criteria	
How does this solve the problem? Yes; as long as we can correctly map function to function, if both implementations do the same, they are the same.	**5**
Does it fit the requirements of the environment? To our knowledge there is no practical implementation of bisimulation for any suitably advanced language. Some proof of concept work for Java exists, but no explicit mention is made of its completeness.	**1**
What is the impact on the development process? (Short and long term) After setup, it should be transparent, only pointing out problems when encountered. Nothing any other supporting tool wouldn't do. It would require similar coding styles between all implementations, because the comparison is based on a strict, abstract representation of the code, that might not deal well with creative uses of language features and such.	**3**
What are the practical risks involved in adopting this solution? Set it and forget it; It doesn't impact the development itself, only interjecting when neccesary. Any problems it presents could be safely ignored.	**5**

Test generation

QuickCheck provides a solid example for the case of test generation. As such, we don't address other available solutions. Those might use different or simpler methods, but we expect this presents a suitable example, with the most significant difference in the difficulty of writing tests. Also, most do not improve on language support. While QuickCheck is useful by itself, we would hope to take advantage of a similarity in function and a broad userspace, to allow testing multiple versions at the same time.

How it addresses the criteria	
How does this solve the problem?	**3**
If setup correctly, it would be able to detect a lot of problems; we should however be aware that the quality of results is purely dependent on the quality and coverage of tests. Furthermore, the tests only relate to themselves. It will note if a test fails or succeeds, and with QuickCheck, for what value, but it doesn't address the where and for the most part why.	
Does it fit the requirements of the environment?	**4**
There are multiple feasible versions of QuickCheck in a host of languages. There are some problems where not all versions use the same syntax, either due to specifics of the language or a liberal interpretation of QuickCheck. It is hard to correctly estimate the impact of differences between language-specific implementations, though the assumption is that those aren't insurmountable.	
What is the impact on the development process? (Short and long term)	**3**
Using QuickCheck could largely supplant the use of other types of unit tests, but would require learning its syntax, as with any external tool. Also, because of the way tests are constructed, if the available generators for test data do not suffice, writing new ones would add considerable complexity to the test process.	
What are the practical risks involved in adopting this solution?	**4**
QuickCheck is a proven technology, even if not all of its implementations are. Furthermore, because it only touches testing, and avoids the program itself, the risk it introduces is limited.	

Static analysis

While this is more of a class of components or approaches we have found no existing tools suited to the tasks we aim to use them for. As such we look more at the suitability of the general principles of static analysis and some related tools instead of putting the focus on the feasibility of a specific product. Furthermore, we will keep to the simpler aspects of static analysis, being cognizant that the specific dynamic nature of JavaScript renders more complex techniques void when applied to the generally accepted forms of JavaScript currently in use. (The use of mature versions of the language that would allow for more advanced tooling is partially addressed in 8.3.2).

How it addresses the criteria	
How does this solve the problem?	**3**
This approach certainly isn't a cure all and there are many aspects that do not lend themselves well to rigid automated analysis. But some do. And those that don't might still provide enough input that we might not be able to steer, but at the very least guide. Not every 'problem' might be a problem, but resolving ambiguity is currently still a human job.	
Does it fit the requirements of the environment?	**4**
As no suitable tools exist, this approach requires custom development, though there is existing tooling to work with both languages on a syntactical level. This reframes the issue to the ability to effectively link those tools and employ them to provide this assistance we require.	
What is the impact on the development process? (Short and long term)	**4**
Due to its nature as supporting tooling, it should have little impact on the development in the short term, beyond hopefully catching potential problems, and allowing them to be solved before those problems present elsewhere. In the long term, the pointers provided by the tools might have trained developers to avoid patterns and techniques that might invoke the wrath of the tool, thus passively improving quality. Depending on the type of analysis performed, issues beyond the original scope of the tool might be identified.	
What are the practical risks involved in adopting this solution?	**4**
As the tool would not be a critical part of the development process, but merely supporting, any problems resulting from it stopping functioning (correctly), could always be resolved by stopping its use. We should recognize that user dependence might have been created, like with spellcheckers.	

9. The first attempt

While the actual approach used will be discussed in chapter 10, this chapter will discus some of the underlying techniques implemented in the tool, and how findings from the first attempt at implementing this functionality informed the design choices in the current version. In working towards a proof of concept, a simple wrapper around the ANTLR parsers for the relevant languages was made. This converted the AST's produced by the ANTLR parsers from the original sourcecode and outputted them in versatile and portable JSON. Then, to see if these JSON AST's met our needs, a few additional scripts were written that provided an elementary wrapper around structure in the JSON files, and performed some elementary operations to extract some basic structural data. This provided some intital security that the basic premise of this project was feasible. Though, due to the purpose of these scripts, there wasn't a lot of care taken with their design and implementation.

First we will describe the AST format as produced by ANTLR. Then we describe the design of the proof of concept scripts. This is followed by a discussion of the problems this design presented, and how these informed the design of the new tool.

9.1. AST format as produced by ANTLR

ANTLR was originally limited to Java, with some recent work on C# and other languages. Still, especially related to some specific grammars, one can not escape the original Java version. As such, an intermediary tool was produced that used the ANTLR grammars and produced an largely platform agnostic, though somewhat verbose, representation in JSON. As we only work with the JSON version, functionality natively provided in the Java representation will not be addressed. Furthermore, not all property values found in the JSON will be explicitly named, if they were not found to be useful for use in processing.

Example of an AST JSON leaf

```json
{
    "text": "executeJqueryNoConflict",
    "token": {
        "channel": 0,
        "charPositionInLine": 0,
        "line": 1,
        "startIndex": 0,
        "stopIndex": 25,
        "text": "executeJqueryNoConflict",
        "tokenIndex": 0,
        "type": 79
    },
    "type": 79
}
```

Nodes consist of a value, a type, a list of children, and a token object containing metadata about the actual text the node represents.

The value can be the actual textual value as contained in the source code or an abstract name for the language construct.

The type contains an id for the type of node that is represented. In the case of the abstract value, the type id generally represents the same distinction.

The token information has no real use for processing the functionality of the code, but does allow you to link specific nodes in the tree back to the source code that specifies them.

As expected in a tree, the list of children contains nodes with the same format, or is empty in case of a leaf node. Similarly, the file itself is represented as the root node.

9.2. Design Proof of concept application

To work with the AST's provided by ANTLR, a naive wrapper object was created that encapsulated the AST of the entire file.

This object, when instantiated with an AST, processes the entire tree breadth first, cycling through all the nodes, making an attempt to process when something interesting comes along.

After any metadata was extracted during the intial processing of the tree, the class still offered rudimentary querying and tree traversal, and a serialization function, so additional information not extracted during the first processing could still be retrieved.

Rather than using a modular, extensible approach, the inner working mostly relied on a few monolithic functions that were ran on every node. These functions would use a select statement with tens of cases to see if the current node would be interesting in some way, and if so to take that node and extract all usable information from it. Due to the nature of this process, any context would mostly be ignored. For the few cases where context was found to be indispensable, the entry point for the interpreter was just shifted to a higher level in the tree. This would result in a situation where, for example, if we needed context to determine variable scope, rather than determining scope when encountering a variable initiation or assignment, by looking for an enclosing block structure, the definition of these scope prescribing structures was amended with the responsibility to process and decorate any scope requiring statements they might contain.

9.3. Where does this go wrong

This worked perfectly fine to extract rudimentary structural data about the different parts of the application, which then could be used to provide some interesting metrics and clarify the relationship between C# and JavaScript implementations. Using this method, some interesting differences in testing approaches and coverage for the C# and JavaScript implementations were found. However, as might be gleamed from the description of the inner workings, it introduced considerable complexity in modifying and extending its functionality. Adding support for some language features would require extensive rewriting of existing code and dealing with more complicated concepts like inheritance and a cross file environment required extensive additional classes to provide a moderately accurate representation of that functionality. A further problem is that the one time, breadth first processing of the file made it largely impossible to extract any useful information that wasn't already explicitly extracted in the processing itself. While that does not pose a problem for the processing itself, it starts to complicate matters when dealing with the results. When processing an AST node in the main loop, the processing was not context aware, so information on parent or sibling nodes would not be available. The only way to include that type of information in the results would be to match an element higher in the tree, and add custom code for the context aware processing of the child node. This created situations where instead of matching on functions and finding the containing element (wether that was the file, an object, or another function), all containing elements would be matched and would try to enumerate all relevant children, so they could be decorated with information about their containing structure. Still, this design required a lot of the logic to be contained outside of the tree representation, and had to make a lot of assumptions about code structure, which created a complex layer of methods and classes, without a clear entry point. Furthermore, this design underestimated the large differences in design for both languages and the resulting divergence in coding style. Not enough care was taken in attempting normalization, but there was also not enough comprehension of the consequences of this paradigm schism. The attempt to provide a general purpose comparator is hindered by the requirement to not only deal with the differences in the languages used, but also the differences in the way both aspects of the tested application are setup and the large part of the application that actually is outside of the scope of this project. (As in, both aspects offer functionality that is in fact not required and thus not included in the other.) This requires a large amount of application specific setup to map the comparable parts of the application and determine scope. This especially applies to tests, where on the C# side, over 70% is without need of an JavaScript equivalent.

9.4. How does this inform the new model

While a far more extensive description of the new tool will be presented in chapter 11, here we will briefly address how the problems discussed in section 9.3 affect its design. The main problem, where the source code is represented as a tree per file, with no specific care paid to its nodes beyond their initial processing is countered by wrapping every node in an object, and building basic traversal and processing functionality on that level. This solves both the maintainability problem of the monolithic select statements, and allows the complete context to be stored on the node level. This creates a case where we can search for every instance of a variable, and then derive the specific context easily, instead of modifying the extraction process itself to decorate the results with this information.

It also creates stability, as, for example, an argument is always represented as an argument, independent of how it was processed or where it was used. Furthermore, we have separated the parsing or processing of the abstract syntax tree and it's normalization. This in the hope to create an abstract version that is suitable for comparison, while maintaining a more language appropriate representation for where that is necessary; this way we can still rely on C# language features like attributes and decorators, while avoiding introducing them in the comparison itself. This also allows us to keep a solid link back to the original source code, which is practically required to allow developers to act on any problems found. This is also a requirement for any attempt to effectively cooperate with a version control system, as you can only reason about the consequences of changes in the source code if you can link those changes to existing and new problems.

10. Technical approach

10.1. Technology choices

The tooling supporting this research is built on existing technology. This section will expand on the reasons for why these technologies were chosen.

10.1.1. ANTLR

ANTLR is probably the most widely known parser generator. As a logical consequence, there are a lot of grammars already available for use with it. This gives us two advantages: not only can we substitute writing two parsers with just writing two grammars, we don't even need to write the grammars, as other people have provided some already.

Though both C# and JavaScript provide a language specification that describes the grammar, actually developing a functional, ANTLR compatible grammar based on that would be non trivial and would add unnecessary overhead, by introducing an additional point of development and potential point of failure.

While we can not ignore that both languages have parsers available that would be more complete or accurate, their output wouldn't be standardized as the one from ANTLR is. This would require additional research if both methods could reach feature parity, and what additional complexity that would introduce. Furthermore, it would require any additional language implementation to write its own mapping between its output and the structure the tool expects, instead of building on the boilerplate included in the tool.

Furthermore, because of ANTLR's wide support we can expect people to be familiar with its workings, which would prove more complicated with other tools.

While an additional advantage was that ANTLR offers multiple compilation targets, in this case we were forced to use Java.

10.1.2. Ruby

The choice for Ruby is mostly a personal one by the author, that ignores some environmental properties and expectations that might suggest some other languages as more suitable.

Beyond personal bias, there were some motivations for this choice. First and foremost, as a dynamic language, it offers a lot of flexibility that allows one to sidestep some of the more rigorous design descisions one would have to make in other languages. However it is still a true objective oriented language, and does prevent many of the problems introduced by fuzzy type systems like those in JavaScript, and lesser, PHP. Furthermore, the language is very expressive, without being verbose. While that might hint to shades of Perl, it generally results in very clear, readable code, which is a good property for a project like this.

While its nature as an interpreted language offers some concerns regarding speed, we expect that speed is a secondary concern, and the advantages of a more performant language would be limited and not merit the additional development effort. Furthermore, there are a number of compilers avaible for Ruby, that offer significant speedups beyond the default interpreter. If speed would become a problem, the tool could be adjusted to work with one of those.

10.2. Tool design

10.2.1. Working with the AST

As mentioned in section 10.1.1, ANTLR only produces Java for use, so the translation from source code to AST is handled by a small Java application that only takes a language argment and filename, and generates a JSON representation of the AST of that file in the same folder using the grammar specified. This requires some additional work, to get arround the memory limit normally imposed on Java processes, because of the length of some of the JavaScript files. JSON was chosen as a fileformat as it is reasonably flexible in the datastructures it can represent, and thus allows mostly effortless serialization, has broad cross-platform support and is both less verbose and complicated than any sort of sufficiently complex XML format.

The resulting JSON files contain a representation of the AST produced by the ANTLR parser, and this datastructure is imported by the Ruby code and then processed recursively, as a tree. Starting with the root node that generally represents the file, every node is tested against the availble node types, and assigned as an instance of a class, based on criteria specified on that class. Matching happens primarily on a first come, first serve basis, which could cause problems if there would be any overlap between criteria for specific classes. This is generally prevented by matching exactly to the nodetype provided by the actual parser, but where this is not sufficient, extra care is taken to make sure the conditions are mututally exclusive.

During this process, every node is assigned a globally unique id, to allow unambiguous reference to specific nodes. While we could defer to passing the node itself during the execution of the program, if we were to decouple the initial processing from further work with the structures, this indirection is neccesary. As one example, during the processing, metadata is saved based on the specific node. If it is a class definition for C#, its name, namespace, and namespaces included in the context are saved. For a method, its class and name are saved. Test are decorated with their designation and a reference to the code they test. And similar, relevant mappings are performed for most types of nodes. Combining this with saving the GUID annotated, cleaned version of the AST in the same way, we can retrieve information about a specific function without requiring parsing and processing every file in the project again.

After the intital encapsulation process is complete, a second sweep is run, that attempts to normalize every node, creating a common representation of the programs between both languages. This removes the distinction between between C# methods and JavaScript functions, and flattens complex stuctures, like those differentiating a literal argument from a member or variable reference. This normalized version exists next to the simple encapsulation, to maintain a clear division between the code that was written and the tool's interpretation. While one might feel this division creates unneccesary duplication, we feel that in its current state the division provides clarity of purpose, explicitly pointing out what representation can be used for what.

Besides this processing step, the class structure offers features for traversing the resulting tree, consisting of simple children_of type methods, but also some more advanced search features, to collect certain nodes from a (sub-)tree. These search features are mostly useful for collecting metadata, especially on a project level.

10.2.2. Storage

The source code itself is stored as-is, accompanied by its JSON AST representation. While this is everything we need, with a decent sized codebase, this would mean processing hundereds of megabytes of inefficiently stored AST data to get the neccesary context for interpreting even the smallest part of the application. Instead we opt for a more efficient storage mechanism, using the files on disk only as a fallback.

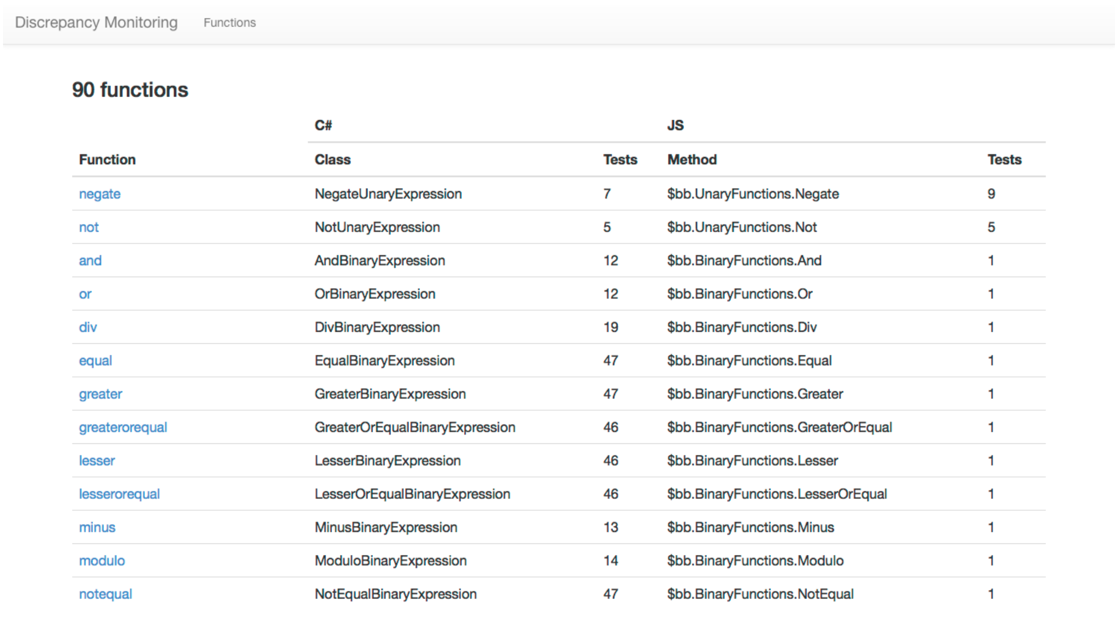
Given that the purpose of this additional storage is to offer a more performant index to the existing information and is completely reconstructable from disk, Redis, a memory based key-value store, was chosen. Not only does it offer broad support for datatypes on the 'value' side of the equation, it also offers basic persistence, to avoid having to rebuild our data after every restart. The primary advantage of a key-value store for this purpose is the lack of a formalized structure. This allows us to use the database as needed, without having to take an ever-expanding datamodel into account, or dealing with complex index and relational structures that might impact both performance and development flexibility.

As mentioned in the previous section, we save both the processed tree and its metadata to Redis, using keys that make it easy to find the relevant segments of code and a datastructure that prevents us both from saving duplicate parts and retrieving segments without context.

While Redis offers some persistence, it is still best considered a volatile datasource. While this meets up with the requirements of what could easily be called a metadata cache.

10.2.3. Reporting

As fixing any problems encountered is beyond the scope of this project, its primary function is getting information about the problems encountered to its users. Due to the scope of the application and in order to keep it stand alone, we do not currently attempt to integrate to other reporting tools or to offer easy integration with, for example, continuous integration tools, especially as we are not aware of any universal format available to provide this functionality, beyond plain text. Instead, we keep reporting contained within the application. The primary interface is offered as a web application, which offers a number of advantages, that are discussed earlier in this document. Primarily, it allows centralizing the tool, without having to rely on an actual client-server model for access. Also, this is far more suitable in a modern development environment than local client software.



90 functions

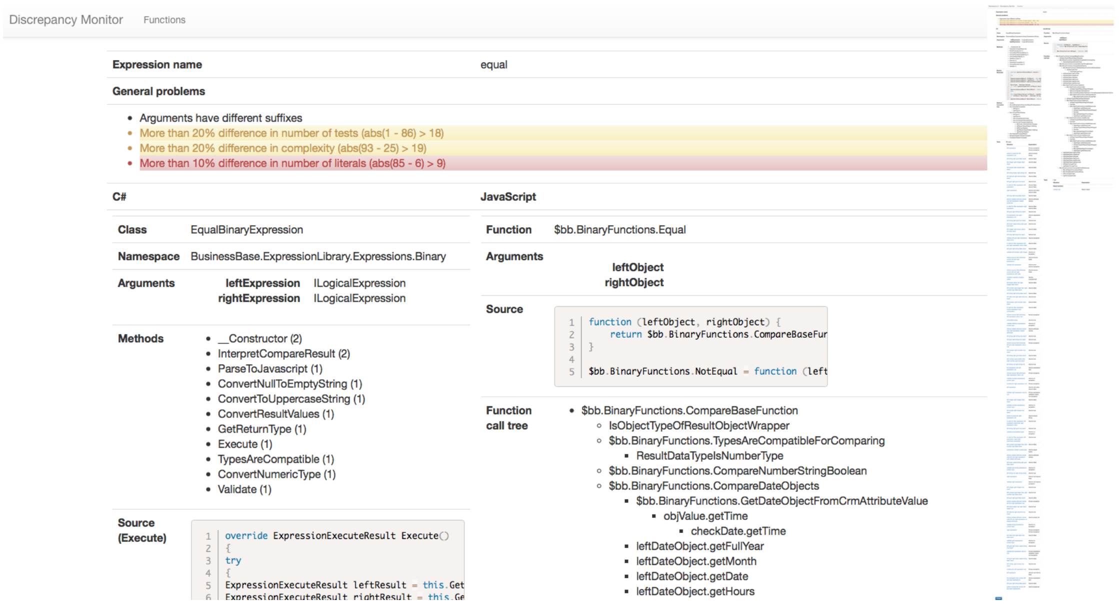
	C#		JS	
Function	Class	Tests	Method	Tests
negate	NegateUnaryExpression	7	$bb.UnaryFunctions.Negate	9
not	NotUnaryExpression	5	$bb.UnaryFunctions.Not	5
and	AndBinaryExpression	12	$bb.BinaryFunctions.And	1
or	OrBinaryExpression	12	$bb.BinaryFunctions.Or	1
div	DivBinaryExpression	19	$bb.BinaryFunctions.Div	1
equal	EqualBinaryExpression	47	$bb.BinaryFunctions.Equal	1
greater	GreaterBinaryExpression	47	$bb.BinaryFunctions.Greater	1
greaterorequal	GreaterOrEqualBinaryExpression	46	$bb.BinaryFunctions.GreaterOrEqual	1
lesser	LesserBinaryExpression	46	$bb.BinaryFunctions.Lesser	1
lesserorequal	LesserOrEqualBinaryExpression	46	$bb.BinaryFunctions.LesserOrEqual	1
minus	MinusBinaryExpression	13	$bb.BinaryFunctions.Minus	1
modulo	ModuloBinaryExpression	14	$bb.BinaryFunctions.Modulo	1
notequal	NotEqualBinaryExpression	47	$bb.BinaryFunctions.NotEqual	1

Figure 10.1.: Function list

The primary function lies in the reporting of potential problems; however the nature of the information collected and the relationships inferred provide better insight in some aspects of the applications architecture than more general tools can provide. As such, while warnings and errors center on actual problems, the general reporting goes beyond that, and attempts to provide a complete picture of the linked code segments and their related (meta)data. In support of this, the start page lists all functions and their respective C# implementing class and JavaScript function, as shown in figure 10.1. This includes views that show definitions of the primitives of the rule language side by side, including (limited) implementation code, and elementary information about test coverage. An example of this view is shown in figure 10.2. We also provide some more indepth information about the tests, allowing a deeper drill down on the tests covering a specific function, as seen in figure 10.3.

10.2.4. Modularity

Learning the lessons from the previous attempt (9) all aspects of the tool use a modular setup, using reflection to automatically extend itself. This keeps specific functional-



Discrepancy Monitor Functions

Expression name equal

General problems

- Arguments have different suffixes
- More than 20% difference in number of tests (abs(1 - 86) > 18)
- More than 20% difference in complexity (abs(93 - 25) > 19)
- More than 10% difference in number of literals (abs(85 - 6) > 9)

C#

Class	EqualBinaryExpression
Namespace	BusinessBase.ExpressionLibrary.Expressions.Binary
Arguments	**leftExpression** ILogicalExpression **rightExpression** ILogicalExpression
Methods	• __Constructor (2) • InterpretCompareResult (2) • ParseToJavascript (1) • ConvertNullToEmptyString (1) • ConvertToUppercaseString (1) • ConvertResultValues (1) • GetReturnType (1) • Execute (1) • TypesAreCompatible (1) • ConvertNumericType (1) • Validate (1)

Source (Execute)

```
override ExpressionExecuteResult Execute()
{
try
{
ExpressionExecuteResult leftResult = this.Get
ExpressionExecuteResult rightResult = this.Ge
```

JavaScript

Function	$bb.BinaryFunctions.Equal
Arguments	**leftObject** **rightObject**

Source

```
function (leftObject, rightObject) {
    return $bb.BinaryFunctions.CompareBaseFur
}

$bb.BinaryFunctions.NotEqual = function (left
```

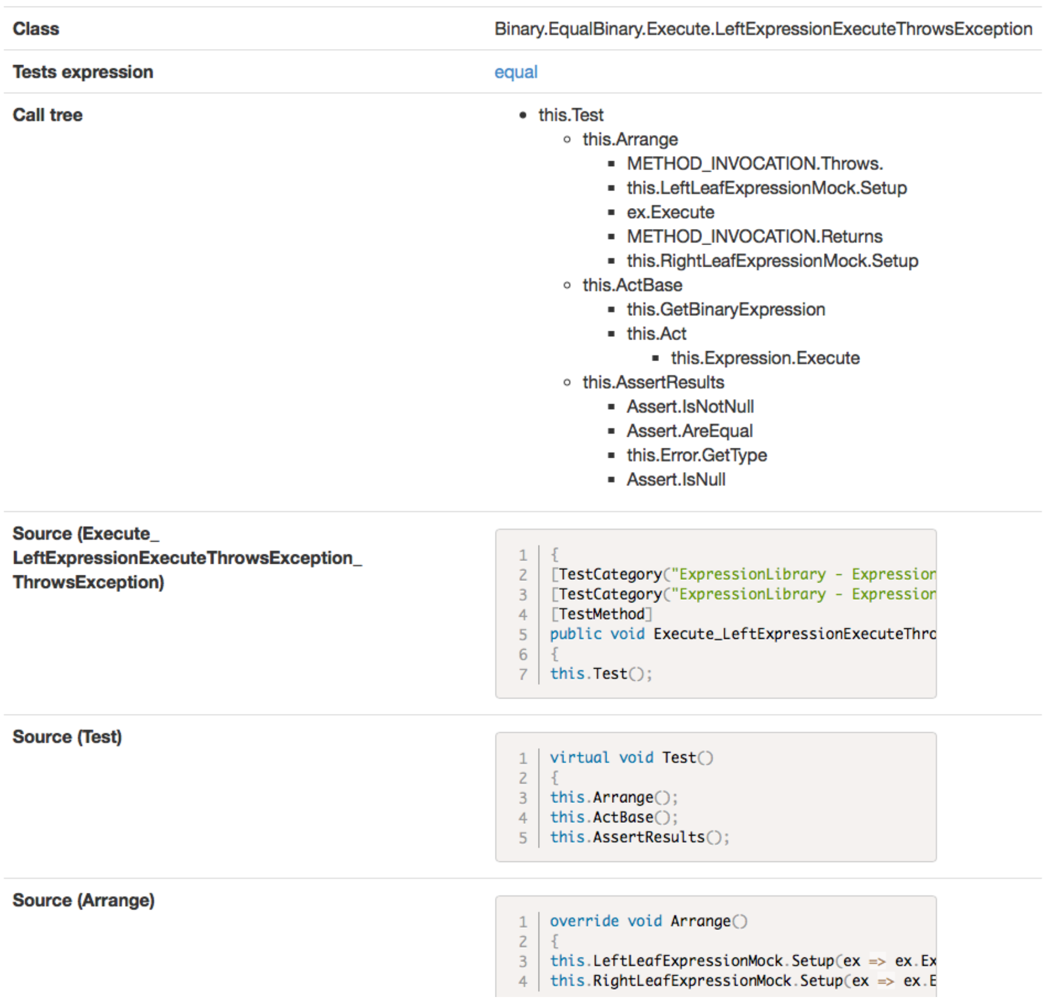
Function call tree

- $bb.BinaryFunctions.CompareBaseFunction
  - IsObjectTypeOfResultObjectWrapper
  - $bb.BinaryFunctions.TypesAreCompatibleForComparing
    - ResultDataTypeIsNumberType
  - $bb.BinaryFunctions.CompareNumberStringBoolean
  - $bb.BinaryFunctions.CompareDateObjects
    - $bb.BinaryFunctions.GetDateObjectFromCrmAttributeValue
      - objValue.getTime
        - checkDate.getTime
    - leftDateObject.getFullYear
    - leftDateObject.getMonth
    - leftDateObject.getDate
    - leftDateObject.getHours

Figure 10.2.: Function detail (with full page length on the right)

lity nicely encapsulated, and because of the reliance on common interfaces, reduces the complexity of extension. While the exact mechanisms themselves aren't important, this allows for great flexibility in our actual testing mechanisms. Instead of relying on a single monolithic operation that attempts to find all ills, we instead have multiple, independent tests for every aspect that is to be judged. This allows for smoother development, as specific attention can be focused on the development of single tasks, their combined functioning being guaranteed by architecture. This furthermore prevents acruing technical debt resulting from scope and feature creep. These tests are discussed in more detail in chapter 11 As in any modular system, attention does need to be paid to the possibility of absence. This becomes especially important with more meta modules, that would (in part) act based on the results of previous modules, as the importance of their reliance on other modules should be weighed. This is to some extent covered by including a rudimentary 'requirements' system, but this does pose a risk to encapsulation.

Modules for testing are allowed to throw messages based on their results, assigning a severity. While generally a specific test will result in a message of a given severity, this does allow some level of scaling when dealing with metrics, as the severity can be determined based on threshold values.



Class	Binary.EqualBinary.Execute.LeftExpressionExecuteThrowsException
Tests expression	equal
Call tree	see below

- this.Test
  - this.Arrange
    - METHOD_INVOCATION.Throws.
    - this.LeftLeafExpressionMock.Setup
    - ex.Execute
    - METHOD_INVOCATION.Returns
    - this.RightLeafExpressionMock.Setup
  - this.ActBase
    - this.GetBinaryExpression
    - this.Act
      - this.Expression.Execute
  - this.AssertResults
    - Assert.IsNotNull
    - Assert.AreEqual
    - this.Error.GetType
    - Assert.IsNull

Source (Execute_ LeftExpressionExecuteThrowsException_ ThrowsException)

```
{
[TestCategory("ExpressionLibrary - Expression
[TestCategory("ExpressionLibrary - Expression
[TestMethod]
public void Execute_LeftExpressionExecuteThro
{
this.Test();
```

Source (Test)

```
virtual void Test()
{
this.Arrange();
this.ActBase();
this.AssertResults();
```

Source (Arrange)

```
override void Arrange()
{
this.LeftLeafExpressionMock.Setup(ex => ex.Ex
this.RightLeafExpressionMock.Setup(ex => ex.E
```

Figure 10.3.: (Above the fold view of a) Test detail page

11. How we do code comparison

As described in section 10.2.1 after the initial processing the tool produces an index of all functions of the 'ExpressionEngine' language, extracted from the metadata. For each of these functions, it can compile a set of the effective executable code for each implementation and related metadata like method signature, related tests, and broader context.

References to the methods combined with their relevant context are inserted into a comparison object, where they are iterated through by a simplified runtime that uses basic matching rules to decide the proper action for a given statement, either an action modifying the current state, or an action introducing one or more branches. At this point we use any available and resolvable state information to decide if specific branches need to be pruned, because they would be unreachable. For these purposes a method/function call is treated as branching with a single branch. We haven't seen any cases where normal execution would go more than about 3 calls deep, so a sane limit was put at 10, to prevent issues with infinite loops.

Beyond the main attraction of the actual code comparison, there are a number of other comparators that focus on less ambiguous characteristics.

11.1. Function signature

Shown in appendix A.2.
Both implementations should be able to handle the same input, so both should have the same signature, with some caveats. C# functions have support for an extra context parameter. This parameter isn't necessary in JavaScript and doesn't represent an input on the original ExpressionEngine expression, so it can safely be ignored. The variable names assigned to the parameters are also compared. We find that to allow better cross implementation understanding, naming should be kept similar. We make special note of pre-, post- or innerfixed variable names, which occur in some cases.

The names used are frequently similar, though especially when some polymorphism is going on, the C# names quickly become generic. E.g. the C# counters for the JavaScript arguments on addseconds (datetime, secondsToAdd) are respectively expressionBaseValue and value, which is far less expressive. A case like todecimal is skipped, because the difference in naming between the JavaScript parameter mainValue and the C# parameter mainExpressionValue is just the inclusion of the 'Expression' string, which happens more often in the C# code.

11.2. Tests

Shown in appendix A.5.
Test coverage for both implementations should be similar, if the difference is significant, this might point to gaps in test coverage or incomplete functionality in one implementation. Furthermore, this module attempts to match tests based on their test cases and expected return values. Ideally both implementations should be covered with the same tests, to avoid missing specific edge cases, or even using incorrect tests. This is limited to a simple match of input and output values. Extra functionality such as Mock testing is not addressed.

A good example of where this proves interesting are the cases of the binary operators (and, or, div, plus, etc.), where the C# implementations all have 40+ tests, covering just about any type of possible input, while the JavaScript versions all make due with just one test to check if something gets returned, with the actual tests for the offered functionality targetted at an 'internal' helper function.

11.3. Cyclomatic complexity

Shown in appendix A.4.
The cyclomatic complexity of both segments is calculated according to McCabe's method [15], and compared. Larger differences in complexity between implementations point to either incomplete implementations or use of library functions. Both cases require additional attention.

The first three sections of chapter 13 provide good examples of methods with wildly differing complexity, primarily due to duplicating library functions from the other implementation.

11.4. External dependencies

Shown in appendix A.6.
While external library functions provide an easy way to delegate responsibility and avoid creating new unique problems, they generally are black boxes, both to the user and tools. Because of this, extra care needs to be taken when using methods outside of the scope of the tool, to ensure they *actually* do what is expected of them.

These are found by scanning for method calls to methods that aren't found in the cache of the tool. This would include calls to Math functions.

While most of these cases are edged towards .NET's extensive default library, and thus can be found by the test from the previous section due to the significant complexity introduced by duplication that functionality, there are some functions that are equaly easy in both languages, and still rely on library functions. One example is capitalize, where both implementations use language provided functions to up- and downcase strings. Though for this specific example there is a case to be made that a case switch is not one's first choice for a library function with large differences between implementations.

11.5. Code comparison

Shown in appendix A.3.
We use a relatively simple process to walk through the code and attempt to find parallels. A number of language features and constructs are unsupported (see section 12.2), so we skip over them. Some of these cases, like Exception handers actually do contain code we want to process, so in those cases the segment is replaced by its contents. We add a special case for 'guard' statements, as these are frequently used in the JavaScript to cover for the lack of a type system, and thus we ignore them. (This applies to conditionals that consist of a single return statement with the value null.)

We walk through the code, looking for the next branching point, either a conditional statement or a method call, while updating the state as we move along. At any branching point, we compare the state between both implementations, then we attempt to match branches by conditon (if there are multiple available) on both implementations, and restart the comparison there.

The functionality that processes branching statements attempts to resolve the conditions, to the extent it is able based on any state filled with literals. This is essential to deal with some compounded case statements for expressions dealing with binary operations. For example, infix operator + is handled by function $bb.BinaryFunctions.Plus, which directly calls $bb.BinaryFunctions.BinaryBaseFunction(leftObject, rightObject, "+"); Based on the "+" string literal, we can prune the case statement in BinaryBaseFunction, thus reducing the complexity found to the actual complexity of the operation, rather the universal case.

12. Language support

Both C# and JavaScript are complex, widely used languages with extensive amounts of features and functionality. C# especially is a very complex language, with a large number of features and a large standard library (generally taking the form of the .NET framework). JavaScript on the other hand is less fully featured, but offers a lot of extensibility by way of its dynamic nature.

This poses a number of problems for any program that attempts to analyze the source code without interpreting. Metaprogramming techniques like reflection can introduce functionality that is in the code but not explicitly defined by the code, requiring interpretation to derive meaning. This, and some other dynamic aspects discussed below, prevent an accurate extraction of control flow without interpretation, and in some cases might even change the program itself during runtime, which renders any attempt to statically derive meaning from source code moot.

We are, however, not attempting a full execution of either program, but are just attempting to find clear markers of potential problems. Because of this we have the freedom to, to a certain extent, pick and choose the functionality that is supported. Furthermore, we take it as a given that both applications are well developed, and we will not have to explicitly look at code quality, 'creative coding' (tricks like the fast inverse square root) or broken code. There is a plethora of very capable tools to help ensure this, that are far more capable than any attempt included in this tool will ever be. This means that we can ignore an entire class of problems introduced by the possibility that the code might not only introduce discrepancies, but could very well be non-functional.

We also have to face the issue that not every feature of one language is available or has a reasonable analogue in the other. For example, the simple, implicit type system from JavaScript can't match the far more complex system offered by C#.

12.1. Properties to explicitly address

12.1.1. Type differences

The most egregious difference is that JavaScript numbers do not address the traditional distinction between floating point, integers and long integers, but are all represented as a number class that is practically a 64-bit float, which gives it an integer precision of about 53 bits. Because this is handled like all numbers are floats, any integer above the 53-bit mark loses resolution, and its value might and probably will differ from the value one supposes it to be. While this is mostly a language problem, which should be addressed in that way, we will attempt to determine by the variable on the other side whether there is a reasonable chance the 53-bit mark might be surpassed.

Other language native types will be addressed, with some attempt being made to take constructs in the JavaScript code that address explicit types into account. More complex types will be ignored for comparison, as discussed in 12.2.1.

12.1.2. Library functions

C# and JavaScript both have library functions that are frequently used directly on literals. As we want to be able to work with some literal values, given that they are frequently used to pick a specific branch within a conditional structure, some simplified implementations of some of the functions encountered will be implemented. While the resulting values will be used to resolve conditions and such, we will avoid processing for cases where we are less than confident about the possible results.

Eg.

```
    var x = '<';
    if (parseStr(x) === '<')
```

is reasonably predictable, but

```
    var t = sprintf("#{0]#{2}--#{1}", "a", otherVar, "c")
    alert("Error:" + t)
```

is far more complicated.

We do make some extra assumptions in extracting meta-data about the application itself, as some required knowledge is stored in more complex method calls. We, however, do not attempt to actually execute the method, but rather use some heuristics on the arguments to attempt to extract the desired information. While this isn't suitable for the general purpose interpretation the tool needs, it serves our purposes in this special case.

12.2. Properties to be ignored or partially implemented

12.2.1. Complex types

Whereas JavaScript offers 7 types (of which 2 describe 'nothing'), C#'s list is somewhat more extensive. Just numeric types account for 12, and then we get bools, enums, arrays, strings and a number of extra types offered by the standard library. Then we find extra features like Nullable and Generics. As most of these types and features lack direct equivalents in JavaScript, it is hard to judge their use. While the existence of these constructs might allow the derivation of some suggestions for its JavaScript equivalent, defining what these should consistently be, is more suitable as additional research than an integral part of the development of this tool.

In most complex cases, because JavaScript largely lacks types, just ignoring them, when there is no clear mapping, is consequence free.

12.2.2. Contracts

Contracts provide an interesting way to help ensure software quality, by defining expectations for a method that it should meet. Support is not native in most languages, and while it is currently included in the .NET framework, JavaScript use requires an external library. Though C# Contracts are used in the projects, and it certainly wouldn't be impossible to judge the JavaScript based on the Contracts defined in the C# equivalent, this is not within the scope of the project. This exclusion merits reevaluation in the case an actual contracts library is included with the JavaScript implementation. The exact shape that support would have is subject for further discussion.

12.2.3. Exception handling

While most languages have some sort of exception handling constructs, including the ones under discussion, their functioning will not be addressed in the system. The primary reason for this being that exceptions should not generally be used for control flow, and any sort of try/catch construct either functions to log any errors occurring within its bounds or suppresses the program ending prematurely. There are some languages where idiomatic use counters this point, like Python, which favors exception handling over pre-emptive checks (ask for forgiveness, rather than permission). This does not counter, however, that this behavior is unexpected in most languages, including the ones addressed here.

Handling of exceptions in a useful way seems impractical due to the complex semantics assigned to them, especially so in a static context. A few examples of this are:

- Not all exceptions will be explicitly thrown in the program code, it might be reasonable to say that most won't be. They will frequently arise from problems with library functions or misuse of language features. This includes problems like type mismatches and invalid input values.
- Exceptions bubble up the call stack. This means that to successfully create a control flow graph, we will not only have to take into account the current state within the object the method was called on (and depending on the paradigms used in development, some global state) but also where it was called from. As instead of returning dutifully to the point where the function or method was called, we could be thrown to any point in the call stack that is encapsulated in a try-catch construct.
- Not every try-catch construct actually responds to every Exception, but they might be scoped to a specific type in their definition. As a result, even when the previously described problem is solved to some extent, we must still be aware of the inheritance of the Exception being thrown to assess which try-catch construct might catch it.

While most, if not all, of these problems are solvable, they would represent a time commitment that isn't commensurate with the additional discrepancy monitoring quality it would provide. We feel that for the most part matching the existence of try-catch constructs in both implementations provides adequate coverage of this functionality. Checking whether it functions correctly could be left to the specific tests implemented for the function being inspected. Furthermore, as noted with the example of idiomatic Python, use of Exceptions differs between languages, even if they are available. As an example, JavaScript tends to be far less eager with throwing exceptions than C# is, preferring a return value on failure, or just failing silently.

12.2.4. Runtime overloading

An object oriented model like C# uses, offers the opportunity to assign a member of a subclass to a variable of its superclass' type. While this precludes the use of any new properties defined by the subclass, it doesn't prevent the use of any overloaded methods on the subclass. This, of course, is as intended. It does present a problem when statically dealing with variables, as the type of the variable is not enough information to pick the correct method for analysis if a call is executed. Because of this it would be necessary to not only remember the type of the variable, but also the type of the object actually assigned. Though this is no impossible feat when the actual creation occurs within the code under analysis, if it is the result of some sort of external data or happens in some external library, we actually can never know the real type of the object referenced by the variable.

While this is an insurmountable problem, the impact to this project is rather small. In the referenced code base, most inter method interaction is performed by primitive or very simple objects. In the current dataset we haven't found any points where overloading could create an unresolvable situation, as the code mostly ignores complex classes beyond other 'expression' classes. If we would encounter other cases, we will treat the object as if it is of the primary type given to the variable.

12.2.5. JavaScript prototype model

While it is easy to mistake JavaScript for a standard procedural language with some object oriented features (like PHP4), it is actually a prototype based object oriented language. Combined with its dynamic nature, this can make object/class definitions very complex, especially when combined with structures like closures (though not the type as defined in SICP [21]). While the support for these features is essential for working with some of the more complex JavaScript libraries, we find that for the current project support of these features can easily be ignored.

12.2.6. Escape sequences

JavaScript and C# both assign special meaning to escape sequences like \n and \t in string literals. Though both refer to the ANSI standard, there are some differences in implementation, but we will treat these sequences as part of the string, rather than attempting to resolve them. While they might introduce some issues, especially with regards to regular expressions, that primarily applies to user input, and those problems are on the interpreter level, and thus out of scope of the tool.

12.2.7. JavaScript magic casting

It is accepted that JavaScript is not what one would call a ‘good language’ (though this is being addressed in newer ECMAScript specifications). One of the clearer ways it demonstrates this is by the way implicit type casting is implemented. While one can see the reasoning behind most of the implicit casts, they create very complex interactions and hard to intuitively predict outcomes. This has resulted in a variety of interesting consequences, with the most fun being contests that use only language constructs to build complex strings or even programs. More serious is the near universal advice to avoid the fuzzy comparators (== and !=), favoring the exact ones (!== and ===). This dismisses an entire language feature, because its results can not be relied upon to work in a predictable way and thus to avoid strange edge cases. Easier to catch, but no less frustrating is that the incorrect way of using some variables, especially when multiple variables interact, can have unintended consequences. Trying to concatenate a string resulting in casting to a number, automatic string to number conversion working as one expects for some cases, but completely failing in some other, seemingly equivalent cases, and fuzzy comparison being in no way transitive.

While an accurate representation of this model is essential to compare the construction and testing of variables to implementations in other languages, we feel an accurate implementation is not achievable and a version with discrepancies with the actual model does more harm than good. As such, only basic support will be implemented, while features like casting arrays and objects to integers will not be addressed.

As these issues are well known in the JavaScript world, any decent linter, like jshint will point out potential problems of this type and will advise the removal of those types of structures. Thus we assume they are not present in the code base processed.

13. Results

This chapter will discus a selection of interesting results produced by the tool, by presenting the interpretation of the tool and a brief comparison of those results with a human analysis of the possible discrepancies. Then, interviews regarding these, and some other results, were presented to and discussed with both the architect and the main developer on this project.

13.1. not

C#

```
return new ExpressionExecuteResult()
{
  ResultType = DataType.Boolean,
  ResultObject = !Convert.ToBoolean(innerExpressionResult.ResultObject)
}
```

JavaScript

```
if (!IsObjectTypeOfResultObjectWrapper(object)) {
    return new ResultObjectWrapper($bb.ResultDataType.Unsupported, null);
}

if (object.ResultObject == null) {
    return new ResultObjectWrapper($bb.ResultDataType.Boolean, null);
}

return new ResultObjectWrapper($bb.ResultDataType.Boolean, !object.
    ResultObject);
```

Tool's conclusions There are 25 to 5 tests, mainly due to testing multiple types of input. A difference in argument names is found, expression v. object. The comparison finds no notable differences.

Interpretation Basically the only success story. Mostly enabled by knowing that ! casts to boolean in JavaScript, so for the comparison, we can ignore the explicit Convert.ToBoolean, and being able to compare ExpressionExecuteResults to ResultObjectWrappers.

13.2. concatenate

C#

```
try
{
  List<string> valuesToConcat = new List<string>();
  foreach (ILogicalExpression expression in this.Expressions)
  {
    string expressionResult = this.
        GetExpressionExecutionResultStringForExpression(expression);
    if (!string.IsNullOrEmpty(expressionResult))
    {
      valuesToConcat.Add(expressionResult);
    }
  }

  return new ExpressionExecuteResult()
  {
    ResultType = DataType.String,
    ResultObject = string.Join(this.Separator, valuesToConcat)
  };
}
catch (BaseSystemException)
{
  throw;
}
catch (Exception exception)
{
  throw new ExpressionExecutionException(this.FullExpressionText,
      exception);
}
```

JavaScript

```
$bb.StringFunctions.Concatenate = function (arrayParam) {
    var returnDataType = $bb.ResultDataType.String;

    if (arrayParam == null) {
        return new ResultObjectWrapper($bb.ResultDataType.Unsupported,
            null);
    }

    var result = $bb.StringFunctions.SeparatedConcatenate(arrayParam, "")
        ;
    return new ResultObjectWrapper(returnDataType, result);
}

$bb.StringFunctions.SeparatedConcatenate = function (arrayParam,
    separator) {
    var resultString = "";

    $j.each(arrayParam, function (index, value) {
        if (IsObjectTypeOfResultObjectWrapper(value) && value.
            ResultObject != null && value.ResultObject !== "") {
```

```
            var resultDataType = value.ResultDataType;
            value = value.ResultObject;

            if (typeof value === "number") {
                var isDecimal = ResultDataTypeIsDecimal(resultDataType);
                value = $bb.SystemFunctionFunctions.FormatNumber(value,
                    isDecimal);
            }

            if ($bb.IsObjectDateType(value)) {
                value = $bb.DateTimeFunctions.ToString(value);
            }

            if ($bb.IsObjectCrmLookupObject(value)) {
                value = $bb.StripBracesFromGuidString(value.id);
            }

            resultString += value + separator;
        }
    });

    if (separator != "") {
        resultString = resultString.slice(0, -1);
    }

    return resultString;
}
```

Tool's conclusions

The difference in naming between the JavaScript parameter arrayParam and the C# parameter Expressions is noted (C# signatures are on the constructor, which isn't shown here.) A large difference in cyclomatic complexity is noted, with 4 for the C# implementation and 949 for the JavaScript implementation. While both implementations contain a loop, both have entirely different contents. The JavaScript implementation includes an conditional call after the loop which is absent from C# Both implementations return an object of type string, but the value is a method call in C# and a variable in JavaScript.

Interpretation

The difference in naming isn't a big deal, but should probably be unified. The large difference in cyclomatic complexity is mostly caused by the better abstracted way to find the string value of the parameters. While the C# addresses this using polymorphism, there really is no reason not to extract this complexity in JavaScript to a StringValue helper function. Currently, just to deal with dates, the javascript repeatedly calls some functions with 20+ complexity, which adds up. Setting aside the mass of conditionals, the JavaScript loop builds the resulting string by concatenating one string at a time, while the C# just builds an list of strings. The tool actually 'found' a small discrepancy here. This slice command is intended to cover for the +*separator* in the last iteration of the loop, but doesn't take separators longer than 1 character into account. Concatenate something with ', ', and that comma will still be sticking at the end. And here we see how the C# implementation does it, by 'join'-ing the List, with the separator. There is no reason for the JavaScript implementation not to work the same way.

13.3. todecimal

C#

```
protected override object ExecuteFunction(ILogicalExpression
    mainExpressionValue)
{
    ExpressionExecuteResult expressionResult = this.
        GetExpressionExecutionResultForExpression(mainExpressionValue);
    if (expressionResult.ResultObject != null)
    {
        decimal? mainValue = this.GetDecimalFromExpressionResult(
            expressionResult.ResultObject, expressionResult.ResultType);
        int? decimalsExpressionValue = this.
            GetExpressionExecutionResultIntegerForExpression(this.
            NumberOfDecimals);

        if (mainValue.HasValue && decimalsExpressionValue.HasValue)
        {
            decimalsExpressionValue = System.Math.Max(0, System.Math.Min
                (28, decimalsExpressionValue.Value));

            return System.Math.Round(mainValue.Value,
                decimalsExpressionValue.Value, MidpointRounding.
                AwayFromZero);
        }
    }

    return null;
}
```

JavaScript

```
$bb.ConversionFunctions.ToDecimal = function (mainValue, numberOfDecimals
    ) {
    if (!IsObjectTypeOfResultObjectWrapper(mainValue) || !
        IsObjectTypeOfResultObjectWrapper(numberOfDecimals)) {
        return new ResultObjectWrapper($bb.ResultDataType.Unsupported,
            null);
    }

    if (mainValue.ResultObject == null || $bb.IsValueValidNumber(
        numberOfDecimals.ResultObject) === false) {
        return new ResultObjectWrapper($bb.ResultDataType.Decimal, null);
    }

    numberOfDecimals.ResultObject = Math.max(0, Math.min(28,
        numberOfDecimals.ResultObject));

    if (mainValue.ResultDataType === $bb.ResultDataType.String) {
        var conversionResult = $bb.ConversionFunctions.
            ConvertStringToDecimal(mainValue.ResultObject);

        if (typeof conversionResult != "number") {
            var errorMessage = "Executing a ToDecimal function went wrong
                . Result is: " + conversionResult;
            $bb.ErrorHandler.AddUserError(errorMessage, $bb.
                CurrentExpressionDefinition.
                FormatExpressionDefinitionForError(), "$bb.
                ConversionFunctions.ToDecimal");

            return new ResultObjectWrapper($bb.ResultDataType.Decimal,
                null);
        }

        return $bb.MathFunctions.Round(new ResultObjectWrapper($bb.
            ResultDataType.Decimal, conversionResult), numberOfDecimals);
    }
    else {
        return $bb.MathFunctions.Round(new ResultObjectWrapper($bb.
            ResultDataType.Decimal, mainValue.ResultObject),
            numberOfDecimals);
    }
}

$bb.MathFunctions.Round = function (paramNumber, decimalPlaces) {
    var roundResult = new ResultObjectWrapper($bb.ResultDataType.Integer,
        0);

    if (!IsObjectTypeOfResultObjectWrapper(paramNumber) || $bb.
        IsValueValidNumber(paramNumber.ResultObject) === false) {
        return roundResult;
    }
    else {
        paramNumber = paramNumber.ResultObject;
    }

    if (!IsObjectTypeOfResultObjectWrapper(decimalPlaces) || $bb.
        IsValueValidNumber(decimalPlaces.ResultObject) === false) {
        decimalPlaces = 0;
    }
    else {
        decimalPlaces = decimalPlaces.ResultObject;
    }

    if (decimalPlaces < 0 || decimalPlaces > 20) {
        return roundResult;
    }

    return $bb.MathFunctions.RoundAwayFromZero(paramNumber, decimalPlaces
        );
}

$bb.MathFunctions.RoundAwayFromZero = function (paramNumber,
    decimalPlaces) {
    var roundAwayFromZeroResult = new ResultObjectWrapper($bb.
        ResultDataType.Integer, 0);

    if ($bb.IsValueValidNumber(paramNumber) === false) {
        return roundAwayFromZeroResult;
    }

    if (decimalPlaces == null || decimalPlaces == "") {
        decimalPlaces = 0;
    }

    if (decimalPlaces < 0 || decimalPlaces > 20) {
        return roundAwayFromZeroResult;
    }

    var sign = 1;
    if (paramNumber < 0) {
        sign = -1;
    }

    var scale = Math.pow(10, decimalPlaces);
    var absoluteValue = Math.abs(paramNumber);
    if (decimalPlaces) {
        absoluteValue = absoluteValue * scale;
    }

    var roundedNumber = Math.floor(absoluteValue + 0.5) * sign;
    if (decimalPlaces) {
        roundAwayFromZeroResult.ResultDataType = $bb.ResultDataType.
            Decimal;
        roundAwayFromZeroResult.ResultObject = (roundedNumber / scale);
    }
    else {
        roundAwayFromZeroResult.ResultDataType = $bb.ResultDataType.
            Integer;
        roundAwayFromZeroResult.ResultObject = roundedNumber;
    }

    return roundAwayFromZeroResult;
}
```

Tool's conclusions The difference in naming between the JavaScript parameter mainValue and the C# parameter mainExpressionValue is skipped, as Expression is frequently inserted in the C# parameter names. The parameter numberOfDecimals is the same in JavaScript and C#. A large difference in cyclomatic complexity is noted, with 6 for the C# implementation and 29 for the JavaScript implementation. Due to numberOfDecimals being variable, nothing can be resolved about result type.

Interpretation There is a large difference in cyclomatic complexity, as C# can rely on a library function, while JavaScript has nothing available that is as customizable, to deal with special cases like the specific rounding type. This results in traversing two functions, each with additional guards, before any calculations get done. Besides limiting the ability to determine the return type, numberOfDecimals provides an additional problem. While both conversion functions limit the number of decimals at 28, the $bb.MathFunctions.RoundAwayFromZero function returns 0 when the decimalPlaces argument is larger than 20. While 28 is the maximum for System.Math.Round, any reasoning for 20 seems suspect, as the 53-bit precision of JavaScript numbers with this implementations, should create problems around 15 decimals.

13.4. addseconds

C#

```
protected override object ExecuteFunction(DateTime mainValue, int value)
{
    return mainValue.AddSeconds(value);
}
```

JavaScript

```
$bb.DateTimeFunctions.AddSeconds = function (datetime, secondsToAdd) {
    var returnDataType = $bb.ResultDataType.DateTime;

    if (!IsObjectTypeOfResultObjectWrapper(datetime) || !
        IsObjectTypeOfResultObjectWrapper(secondsToAdd)) {
        return new ResultObjectWrapper($bb.ResultDataType.Unsupported,
            null);
    }

    if (datetime.ResultObject == null || secondsToAdd.ResultObject ==
        null) {
        return new ResultObjectWrapper(returnDataType, null);
    }

    secondsToAdd = Number(secondsToAdd.ResultObject);

    var millisecondsToAddWrapped = new ResultObjectWrapper($bb.
        ResultDataType.Integer, secondsToAdd * 1000);
    var returnValue = $bb.DateTimeFunctions.AddMilliseconds(datetime,
        millisecondsToAddWrapped);
    return new ResultObjectWrapper(returnDataType, returnValue.
        ResultObject);
}

$bb.DateTimeFunctions.AddMilliseconds = function (datetime,
    millisecondsToAdd) {
    var returnDataType = $bb.ResultDataType.DateTime;

    if (!IsObjectTypeOfResultObjectWrapper(datetime) || !
        IsObjectTypeOfResultObjectWrapper(millisecondsToAdd)) {
        return new ResultObjectWrapper($bb.ResultDataType.Unsupported,
            null);
    }

    if (datetime.ResultObject == null || millisecondsToAdd.ResultObject
        == null) {
```

```
        return new ResultObjectWrapper(returnDataType , null);
    }

    millisecondsToAdd = Number(millisecondsToAdd.ResultObject);

    var dateObject = $bb.GetDateObjectFromObject(datetime.ResultObject);

    dateObject.setMilliseconds(dateObject.getMilliseconds() +
        millisecondsToAdd);

    return new ResultObjectWrapper(returnDataType , dateObject);
}
```

Tool's conclusions The difference in naming between the JavaScript parameters datetime and secondsToAdd and the C# parameters expressionBaseValue and value is noted (C# signatures are on the constructor, which isn't shown here.) A large difference in cyclomatic complexity is noted, with 1 for the C# implementation and 5 for the JavaScript implementation. One is literally one statement and the other two functions. Both return a DateTime object. **Interpretation** The parameter names in C# here are very generic, and should be changed to better reflect what values they actually contain, both on the class and the ExecuteFunction method. We find here again the issue with the library call. Lacking a solid data library, what requires one call on a C# DateTime object, requires a detour calculating and offsetting microseconds through an additional function. While no problems seem to be introduced here, the code is tremendously complex for what it aims to do. It would be practical to find a DateTime wrapper for JavaScript that offers similar methods to alter a date as the C# version does. Especially as dates can get complex when dealing with more than adding seconds.

13.5. if

C#

```
public override ExpressionExecuteResult Execute()
{
    ExpressionExecuteResult booleanExpressionResult = (
        ExpressionExecuteResult)this.BooleanExpression.Execute();
    if (booleanExpressionResult == null || booleanExpressionResult.
        ResultObject == null)
    {
        throw new ExpressionExecuteResultNullException(this.
            FullExpressionText);
    }

    if (booleanExpressionResult.ResultType != DataType.Boolean)
    {
        throw new ExpressionResultTypeWrongDataTypeException(this.
            FullExpressionText , new List<DataType> { DataType.Boolean },
            booleanExpressionResult.ResultType);
    }

    ExpressionExecuteResult finalResult = new ExpressionExecuteResult();
    try
    {
        if (Convert.ToBoolean(booleanExpressionResult.ResultObject))
        {
            finalResult = this.TrueExpression.Execute();
        }
        else
        {
            finalResult = this.FalseExpression.Execute();
        }
    }
    catch (BaseSystemException)
    {
        throw;
    }
    catch (Exception exception)
    {
        throw new ExpressionExecutionException("Unhandled exception in "
            + this.FullExpressionText, exception);
    }

    if (finalResult == null)
    {
        throw new ExpressionExecuteResultNullException(this.
            FullExpressionText);
    }

    return finalResult;
}
```

JavaScript

```
$bb.SystemFunctionFunctions.If = function (booleanExpression,
    trueExpression, falseExpression) {
    var resultNull = new ResultObjectWrapper($bb.ResultDataType.
        Unsupported, null);

    if (booleanExpression == null || !IsObjectTypeOfResultObjectWrapper(
        booleanExpression) || booleanExpression.ResultObject == null) {
        return resultNull;
    }

    if (trueExpression == null || !IsObjectTypeOfResultObjectWrapper(
        trueExpression)) {
        return resultNull;
    }

    if (falseExpression == null || !IsObjectTypeOfResultObjectWrapper(
        falseExpression)) {
        return resultNull;
    }

    if (booleanExpression.ResultObject) {
        return trueExpression;
    }
    else {
        return falseExpression;
    }
}
```

Tool's conclusions The parameters for both implementations are the same. No difference in cyclomatic complexity is noted, with 6 for both implementations. Both implementations return variables. The type of the return variable is unclear. **Interpretation** While the complexity is the same, this is for different reasons, the first two conditions check the validity of the booleanExpression, whereas this is just one check in JavaScript. Then, based on the booleanExpression, either the true or false expression is executed, and only then is checked if the result is null. In JavaScript this is done directly after checking the booleanExpression. One of the reasons this is possible points to another problem presented by the JavaScript version of this function. As the entire expression is send to the browser as an 'eval'-able string, both the true and false branch are executed to be able to call the If function with the relevant arguments. While this generally doesn't pose a functional problem, as the functions don't have side effects, there is a performance penalty as any expression on either branch would need the be executed, even if that expression was dependent on an call to a server, or the processing of other fields on a page. This would be hard to fix, without replacing the current generated JavaScript from a function call to something like this, making the generated code slightly more complex.

13.6. Use for development

A selection of the results in the previous sections, combined with some omitted here, were discussed with the two development team members that are actively involved with the affected application.

13.6.1. Architect

The architect of the application isn't very surprised by the nature of some of the issues found. Nor by the limits of the tool. JavaScript development had never been anyones specific skill, and when something met its functional requirements, had decent coverage, and managed to pass those tests, that was enough till problems emerged. Mitigating the lack of equivalent library features across languages wasn't addressed by finding other libraries that might fill those voids, as bad interoperability with the existing environment was feared. Any actual issues found from tests with such an approach weren't mentioned. Though the type of discrepancies the tool was able to point out were found to be of interest, the question did arise if the realization of a tool like this would be cost effective without the research aspect. This is where it was reasoned that, especially given the scale of the monitored codebase, and the presented limitations, most of the same advantages could be achieved by modifying the process. By placing more importance on a singular source representation, that would then be mirrored as closely as possible in the other language, which would avoid situations with functionally similar but conceptually different solutions, like the one in the concatenate function above. An continuing interest in the opportunities presented by test generation was expressed also, though a slightly deeper dive on the subject also surfaced the prominent chance that this technique would have a similar limitation in an askew ratio between setup and resolved issues. Finally the conversation moved to the more general issue of finding a sweet spot in supporting tooling, where the effort spent on setup and upkeep doesn't overtake either the technical or business results.

13.6.2. Developer

The developer responsible for (close to) all JavaScript code was quick to qualify some of the issues that were found by noting that he isn't primarily a JavaScript developer. Also that, especially with early code, there was a big focus on making it work, over being well designed. A further look at the type of issues that were found led to some interesting insights in the underlying design process. Like, again, the concatenate function in section 13.2; while there is definitely a discrepancy there, it wouldn't be triggered in any real use case, as the '$bb.StringFunctions.SeparatedConcatenate' function only got called from other functions with an at most 1 character long separator argument. Which presents an interesting issue with building a function to meet a rigid functional description. When discussing what type of improvements are suggested by the basic results of the tool, the discussion quickly falls back to detecting outliers, issues with greatly differing complexity or other more obvious signals, as a road map for inspection and improvement. When focusing on how we get from the detection to actually producing less error prone, or at least more maintainable code, he is apprehensive of more general approaches, given the effort involved and some environmental conditions, like language specific features. While going more in dept, he can see advantages of attempting to structure both segments of code in comparable ways, though still reserving some doubts with regards to library functions and language constructs. Most of these suggestions and approaches were quickly scoped with feasibility in a business context, but also the specific constraints of the existing architecture.

13.6.3. Summary

Interestingly enough, one of the more important issues in both interviews was feasibility, both with regards to the existing architecture and the business case. Both dismissed options or scoped decisions within limits posed by the existing architecture, and both addressed the feasibility of acting on the results of this analysis. Though in the case of the architect (who also is the CTO), this focused more on the meta issue of changing the development environment and process, while the developer focused on the actual impact of executing some of the changes that were suggested. This reinforces that in professional software development, at a certain size of issue, a solution needs both a technical and business axis. When looking at the results itself, there is little surprise that some of the approaches chosen are suboptimal; with working code taking precedent over design. It is also readily acknowledged that most of the issues that have been encountered in the past fall outside of the scope of a decent comparison as it generally turns on mismatches in available functionality in the default libraries. An agreement could be reached that the best way to prevent future issues would be reducing the opportunity for discrepancies by putting a stronger focus on easily comparable implementations, though this does nothing for existing code and the problems that might lie within. This would be the use case where the tool could be used to pinpoint cases that might need a second look based on more generic comparisons, like comparable complexity.

Practical impact

At this time the practical impact of the research at the case company is very limited. Most of the problems detected in actual implementations are not specific, actionable issues but primarily broader issues with structure, which brings us to the feasibility of rewriting large parts of a working application for mostly theoretical gains. While this would ideally improve the ability of the tool to analyze the code, and simultaneously result in code that is easier to understand and compare to their other implementation, it is hard to accurately predict any measurable improvements resulting from what would certainly be a huge undertaking. Some meta-results did lead to action, though actual impact is still in doubt. An earlier discussion of the discrepancies in test coverage between implementations (as mentioned in section 11.2) was acknowledged and supposed to be acted upon, though at a recent check no real evidence of progress in that case could be found. Based on analysis of the JavaScript expression engine runtime, justified in part by issues like the one regarding if (section 13.5), we were asked to implement a proof-of-concept of a new runtime that would avoid these issues. While the proof-of-concept performed well enough to justify putting this rewrite on the road map, the start date has been postponed multiple time, and as of this writing there is no clear start date for the project.

14. Discussion

This thesis adresses a pretty hard problem and the results show this. While, till about the architectural level, you are bound by convention and context, and it is easy to derive structure and relationships, once you get close enough to the functionality itself, the semantics become muddled. Not line by line, but the interplay.

Many actions could easily be reduced to a very small number of independently very understandable commands (it is not rare to see code standards limit functions to 15-25 lines), but the actual functionality derives from their interaction, which isn't always as clear. The nicest example of this might be the XOR-swap algorithm. While rudimentary knowledge of binary math shows that $x \coloneqq x \oplus y; y \coloneqq x \oplus y; x \coloneqq x \oplus y$ and $z \coloneqq y; y \coloneqq x; x \coloneqq z$ both end with the values of x and y swapped, to do it in the context of automated, static analysis, you would either need to introduce it as a pattern (where you would have to address that it doesn't work for complex types), or any rudimentary understanding of the process would put you three quarters towards a (very simple) interpreter.

And this example obviously isn't unique. Every problem in software engineering has a million different solutions. Which means we quickly end up with a tool that doesn't tell if two pieces of code do the same thing, but merely if they do it in the same way. That is still useful information, though. We probably want things to be done the same way within the same application. But at the same time, it risks creating a boy who cried wolf scenario, where it becomes easy to dismiss potential problems under the guise of the tool detecting only the differing implementation, instead of the underlying bugs, especially when the relevant code does pass the test suite. A good example of this is the concatenation function addressed in chapter 13, where the tool found both implementations to use sufficiently different methods, but couldn't address the actual discrepancy arising from .slice in the JavaScript implementation.

But when we look at some of the other problems that are identified, this 'do repeat yourself' mantra is less suitable. While unifying the differing approaches from above makes sense, applying the same to a case where one function is implemented by a simple library call and the other is implemented in full, would require introducing new complexity for the sake of a better ability to detect potential problems. While there are numerous cases for and against adopting a 'Not Invented Here' mentality, it's hard to enforce such a hard-line when dealing with complex concepts like dates, regular expressions, or Unicode. And even some concepts with converting numbers. Like the decimal string parsing function that gets tagged in JavaScript, because it is tremendously more complex, actually detecting however, that the JavaScript round function caps decimals at 20, while the conversion functions in both languages explicitly use 28, is not going further than noting the use of library in C#;

At the end of the day, tools remain tools, and can only *assist* people in resolving or preventing problems. Whereas a more involved tool might embed itself into the development process to such an extent that it removes agency and responsibility from its users, by eliminating some friction or reducing some complexity, showing instant, measurable results in productivity, tools like this have a longer tail, hopefully trading the lack of short term gain for long term gradual improvement.

15. Conclusion

The past few pages have attempted to address the following question *How can we prevent introducing discrepancies in functionality between two similar implementations in distinct programming languages.* And in doing so we find both the cause and the solution in the same place as with most technological problems. People.

Discrepancies mainly arise from incomplete specifications and bad programming. The language might not always prove the most helpful ally, but problems from silly corner-cases are rarer than one might initially expect. Instead, problems come from leaving assumptions about rounding open, not specifying the exact conditions for a value to be correct, or not distinguishing two similar arguments for a function.

Most of the more effective techniques to prevent discrepancies just eliminate the need for any (written) duplicated code, either by using JavaScript server side, generating the JavaScript from the server-side code or just doing all the processing server side using (micro-)services. These are all fair approaches for new projects. All the approaches that are more amenable for a drop-in solution, either offer little advantage, have even more esoteric requirements of the environment, or are simply too complex to implement. No silver bullet here. A custom static analysis based approach had the best risk-reward ratio, mostly because the other approaches were found to be infeasible.

The impact of good process can not be overstated. While tools of the sort we have discussed can find problems, and might even suggest solutions, process is the only way to actively prevent the introduction of discrepancies. This is achieved by a multi-pronged approach that includes not only defining requirements on a functional level, but also on an implementation or algorithm level. Furthermore, we require that in code review, at some point, both implementations are checked against each other. Also, we require that test cases are synchronized between implementations.

The type of discrepancies we are able to detect lacks a direct parallel to their origins, and is (not too surprising) related to their implementation. While we are able to point out problematic uses of functions, or situations that could easily present subtle differences in functioning, concrete results require actual differences in implementation. This mostly makes it impossible to detect the most insidious type of discrepancies, those emerging from edge cases in libraries. Furthermore, we find that being able to accurately deal with different coding patterns emerging from language features requires either active participation by the developers to eschew idiomatic language constructs in favor easily duplicated ones or extensive heuristics that attempt to interpret equivalence of intent. Neither of those lends itself well to interaction with existing code.

There is no general purpose, fool-proof way to ensure no discrepancies are introduced in an application that isn't DRY, within the parameters of most development projects, however, we can mitigate these problems with improvements in process and tooling. By ensuring developers are acutely aware that their environment is out to introduce bugs into their code, an understanding of problematic patterns and behaviors will emerge. We can support and shape this understanding with tools, definitely not just limited to a discrepancy monitor, but also including project management, testing, linting, style guides and documentation, eventually resulting more effective development.

16. Future research

As mentioned a few times, most prominently in 5.1, this research, and its accompanying proof of concept, are intended as a starting point.

In this chapter we will discuss a number of opportunities for future reasearch, based our current results.

16.1. Practical solutions

Chapter 8 addresses the feasibility of a number of solutions for our problem of code duplication. While the intent was to provide an overview and evaluation that would be universally applicable, there is opportunity for improvement.

In the selection of possible solutions we were to some extent guided by the requirements of the case company. Though we did take care this had little impact on the breath of solutions considered, we avoided going in depth on possible solutions that showed an immediate mismatch with the environment at the case company. This means that there are a number, possibly quite large, of potential solutions that were either dismissed out of hand or only quickly touched upon, while they might provide a feasible solution in some other circumstances. Extending this overview of possible solutions with choices based either on different circumstances or environment agnostic would be an interesting opportunity.

Similarily, the selection of solutions was based on a case where the technologies in the environment were set and an codebase exists. There is an opportunity to take a look at the problem from a tabula rasa, where the technology is chosen based on the requirements, instead of vice versa. Especially given the current developments in this area, both with regards to newer frameworks and paradigms that eliminate duplicated codebases and better support for using JavaScript as a compilation target.

16.2. Tool quality

These is a number of design decisions made in developing the proof of concept tooling for this project that could benefit from some reexamination. As these choices were made to aid in the flexibility of the development, and to provide a clear, rather than effective structure, a closer examination of the tool would provide numerous points of improvement. To present a few examples; the hard division between the ANTLR parser and tool required by the use of ANLTR and Java-based grammars is at least somewhat problematic. Given the extensive number of languages supported in ANTLR 3, one could rework the grammars and move the entire program to a single language, perhaps C#.

Though the division between the direct and normalized respresentations is essential, implementation as seperate 'Parser', 'Normalisation' and even 'Runtime' classes isn't going to win any beauty pageants, either, though it represents the structure of the program well. Ideally this division would be refactored out in some way, though the exact form this restructuring would take is left as an exercise to the reader.

Furthermore, while the current code focusses normalization specifically on independant statements, in some cases broader structures are interpreted. Better heuristics on this point might allow reduction of methods to a more abstract representation that is more suited to comparison.

16.3. Reevalution of tools

Especially on the C# side, a lot of progress has been made with the availability of low level development tools, because of a push by Microsoft to focus more on infrastructure, rather than COTS software products. A cursory investigation of the current state-of-the-art shows no significant improvements, though a more in-depth look might unearth relevant improvements. Similarly, JavaScript progress is always substantial, so some of the decisions there could benefit from reevaluation.

16.4. Feature completeness

Chapter 12 describes a number of (language) features that were either implemented partially or omitted completely. While we find that some of these omissions present insurmountable problems that represent theoretical limitations of the chosen approach, the majority is excluded based on the scope of the research, the requirements of the specific case, assumptions about the use of the tool and potential conflicts with other tools.

As such, the subsections of this chapter present well delineated opportunities for further research. Beyond the more direct approach of patching holes, this also points to interesting complexities of the involved languages that could form the basis a less directly connected project. For example, there have been attempts use JavaScripts dynamic nature and casting system to find a minimum character subset that offers full functionality, or somewhat more practical, a way to visualize exception handling.

16.5. Language support

This project only implements the comparison for two languages, JavaScript and C#. While the comparison modules should function on any tuple processed abstract syntax trees, this has not been tested, and really can't be tested unless other implementations of the language specific parts of the tool are produced. It would be interesting to see the effort required to add other languages to those supported by the tool, and if the functionality they offer can be mapped to the available framework or if it would require extensions or additional mappings.

16.6. Project agnosticism

The current implementation uses a number of additional classes and methods to map C# classes to the equivalent JavaScript functions, based on (meta)data available in the source code. Especially given the mapping of complete classes on the C# side to simple functions in JavaScript, there really seems to be no fool proof way to eliminate this custom setup code, for this specific case, but we expect that for most projects, if there is a more direct link between the implementations in both languages, that even mapping itself should be able to be derived, based on information like method signatures.

We feel that this would probably be an interesting tool, both as a standalone to compare API's or services (which might very well already exist), or as a part of the tool from this project, as a way to prevent or eliminate project specific code.

16.7. Reporting and documentation

Given that the tool does not limit itself only to operational code for its comparisons, it gathers extensive amounts of meta data about method signatures, software architecture, test cases and more. If the tested code, as it is here, is intended to be user facing, this sort of information could form a baseline for generating documentation. If we were to actually decorate the meta data as it is generated by the tool, a variation of its current reporting functionality could provide a self updating reference manual, constructing examples from unit tests and marking when the documentation goes out of sync with the underlying code.

16.8. Long term impact

As discussed most extensively in chapter 14, we expect the use of a tool of this type to have little immediate effect on the development process, but effect a more gradual improvement in code quality over a longer timespan. As this belief comes mostly from anecdotal data from the authors professional experience, a more thorough examination of effect could be warranted. This might be hindered by most programmers already self-reporting monumental improvements in skill over longer time spans. (The well-known “I don’t know who wrote this horrible code... Oh, it was me, 18 months ago” issue.)

A. Source for testers

This appendix includes the sourcecode for the testers used in the DiscrepancyMonitor tool

A.1. Test baseclass and factory

```ruby
module DiscrepancyMonitor
  module Tester
    class Test
      class Tests
        def initialize
          @list = []
          @types = %i(meta source state type)
        end

        def add(item)
          @list << item
        end

        def list
          if @typed_list.nil?
            @typed_list = {}
            @list.each do |klass|
              check_class_type(klass)
              @typed_list[klass.type] = [] unless @typed_list.key? klass.
                  type
              @typed_list[klass.type] << klass
            end
            @typed_list = @typed_list.map do |t, c|
              [t, c.sort_by(&:weight)]
            end.to_h
          end
          @typed_list
        end

        def for(type)
          return [] unless list.key? type
          list[type]
        end

        def to_a
          list.keys.to_a
        end

        protected

    def check_class_type(klass)
      return if @types.include?(klass.type)
      error = 'Attempted to add test @test with unknown type @type'
      error = error.gsub('@test', klass.to_s)
                   .gsub('@type', klass.type.to_s)
      throw Exception.new(error)
    end
  end

  @tests = Tests.new

  def self.tests
    if @tests.nil?
      superclass.tests
    else
      @tests
    end
  end

  def self.for(type)
    tests.for(type)
  end

  def self.inherited(test)
    tests.add(test)
  end

  def self.type
    nil
  end

  def self.weight
    0
  end

  def self.run(type, data, pair)
    case type
    when :meta, :source, :state
      run_tasks(type, data, pair)
    end
  end

  def self.run_tasks(type, data, pair)
    self.for(type).each do |test_class|
      test = test_class.new(data, pair)
      test.run
    end
  end

  def initialize(data, pair)
    @data = data
    @pair = pair
  end
```

```

      def name
        :base
      end

      def add_note(text, type = :notice)
        @pair.add_note(text, type)
      end

      def run
        throw Exception.new('You should implement this!')
      end

      def set_state(field, value, test_name = nil)
        test_name ||= name
        @pair.test_state[test_name] = {} unless @pair.test_state.key?
            test_name
        @pair.test_state[test_name][field] = value
      end

      def get_state(field, test_name = nil)
        test_name ||= name
        begin
          return @pair.test_state[test_name][field]
        rescue
          return nil
        end
      end

    end

    class TestFactory
      def initialize(pair)
        @pair = pair
      end

      def run(type, data)
        Test.run(type, data, @pair)
      end
    end
  end
end
```

A.2. Arguments

```
module DiscrepancyMonitor
  module Tester
    class Arguments < Test
      CS_ARG_TYPES = %w(IContext ILogicalExpression ILogicalExpression
          .[]).freeze
      ADDITIONS_EX = /Value|Object|Expression/

      def self.type
```

```
        :meta
      end

      def name
        :arguments
      end

      def run
        @args = @data.map { |l, v| [l, v[:args].clone] }.to_h
        check_and_clear_csharp_types
        check_number_of_args
        check_additions
      end

      protected

      def check_number_of_args
        return if @args.values.map(&:size).uniq.length == 1
        add_note('Different number of arguments for both implementations'
            , :danger)
      end

      def check_and_clear_csharp_types
        check_csharp_types
        @args[:csharp].reject! { |arg| arg[:type] == 'IContext' }
        @args[:csharp].map! { |arg| arg[:name] }
      end

      def check_csharp_types
        @args[:csharp].each do |arg|
          next if CS_ARG_TYPES.include?(arg[:type])
          add_note('C# implementation has an argument that isn\'t a
              context or expression')
        end
      end

      def check_additions
        return if @args.values.reduce(&:==)
        args = @args.map do |_, v|
          v.map { |arg| arg.gsub(ADDITIONS_EX, '') }
        end
        unless args.reduce(&:==)
          add_note('Argument names for both implementations differ', :
              warning)
          return
        end
        add_note('Arguments have different suffixes')
      end
    end
  end
end
```

A.3. Compare

```ruby
module DiscrepancyMonitor
  module Tester
    class Compare < Test
      def self.type
        :state
      end

      def name
        :compare
      end

      def run
        do_comparison
      end

      protected

      def do_comparison
        order_data
        compare_actions
      end

      def order_data
        @data = @data.sort_by(&:length)
      end

      def compare_actions
        @data[0].actions.product(@data[1].actions).each do |s|
          s[0].add_equal(s[1]) if s[0].equal?(s[1])
        end
        @results = @data.map do |branch|
          [branch.language, branch.actions.reject(&:matched?)]
        end.to_h
        @results.each do |l, v|
          unless v.length.zero?
            add_note('Found ' + v.length.to_s + ' unmatched ' + l.to_s +
                ' statements', :console)
          end
        end
      end
    end
  end
end
```

A.4. Complexity

```ruby
module DiscrepancyMonitor
  module Tester
    class Complexity < Test
      include MetricWithLimit

    def self.type
      :source
    end

    def name
      :complexity
    end

    def run
      add_large_diff_note('complexity', :warning) if diff > limit
    end

    protected

    def amounts
      {
        js: calc_complexity_js,
        csharp: calc_complexity_csharp
      }
    end

    def limit_val
      0.2
    end

    def calc_complexity_csharp
      calc_complexity([
        conditional_branches_csharp,
        loops_csharp
      ])
    end

    def conditional_branches_csharp
      conditional_branches([
        Parser::CSharpIf,
        Parser::CSharpSwitch
      ], @data[:csharp])
    end

    def loops_csharp
      loops([
        Parser::CSharpForEachLoop,
        Parser::CSharpForLoop
      ], @data[:csharp])
    end

    def calc_complexity_js
      calc_complexity([
        conditional_branches_js,
        loops_js,
        faux_loops_js
      ])
    end

```

```
      def conditional_branches_js
        conditional_branches([
          Parser::JavascriptTernary,
          Parser::JavascriptSwitch,
          Parser::JavascriptIf
        ], @data[:js])
      end

      def loops_js
        loops([Parser::JavascriptForLoop], @data[:js])
      end

      def calc_complexity(list)
        list << 1
        list.compact.reduce(:+)
      end

      # We ignore short-circuits
      # Also: cases are relevant edges, else just continues
      def conditional_branches(segment_types, base_segment)
        segment_types.map do |type|
          base_segment.select_deep(type).map do |segment|
            segment.cases.length
          end.compact.reduce(:+)
        end.compact.reduce(:+)
      end

      def loops(segment_types, base_segment)
        segment_types.map do |type|
          base_segment.select_deep(type).length
        end.compact.reduce(:+)
      end

      def normalize_jquery_naming(method)
        method.gsub('$j.', 'jQuery.').gsub('$.', 'jQuery.')
      end

      def faux_loops_js
        loop_methods = %w(jQuery.each jQuery.map)
        [Parser::JavascriptCall].map do |type|
          @data[:js].select_deep(type).map do |segment|
            method = normalize_jquery_naming(segment.method.text)
            loop_methods.include?(method) || nil
          end.compact.length
        end.compact.reduce(:+)
      end
    end
  end
end
```

A.5. Tests

```
module DiscrepancyMonitor
```

```ruby
  module Tester
    class EqTests < Test
      include MetricWithLimit

      def self.type
        :meta
      end

      def name
        :eq_tests
      end

      def run
        if max.zero?
          add_note('No tests found', :danger)
        elsif diff < limit
          add_note('Small difference in number of tests', :info)
        else
          add_large_diff_note('number of tests', :warning)
        end
      end

      protected

      def limit_val
        0.2
      end

      def amounts
        @amounts ||= {
          js: @data[:js][:tests].values.flatten.length,
          csharp: @data[:csharp][:tests].length
        }
      end
    end
  end
end
```

A.6. Libraries

```ruby
module DiscrepancyMonitor
  module Tester
    class Libraries < Test
      include MetricWithLimit

      def self.type
        :source
      end

      def name
        :libraries
      end

      def run
        add_large_diff_note('number of library calls', :warning) if diff > limit
        # find_library_calls # should probably find a better way to display this info
      end

      protected

      def amounts
        {
          js: find_js_library_calls.length,
          csharp: find_cs_library_calls.length
        }
      end

      def find_library_calls
        find_cs_library_calls
        find_js_library_calls
      end

      def find_cs_library_calls
        @cs_calls ||= @data[:csharp].select_deep(Parser::CSharpCall).map do |call|
          name = call.normalized.method.text
          name = name.split('.') unless name.class == Array
          next nil if name[0] == 'this' && name.length == 2
          add_note('Potential CSharp library call: ' + name.join('.'), :console)
        end.compact
      end

      def find_js_library_calls
        @js_calls ||= @data[:js].select_deep(Parser::JavascriptCall).map do |call|
          next if call.method.class == Parser::JavascriptNew
          call_to = call.method.text
          segment = @pair.state[:js].record.get_segment([:method, call_to])
          next nil unless segment.nil?
          add_note('Potential js library call: ' + call_to, :console)
        end.compact
      end
    end
  end
end
```

A.7. Literals

```
module DiscrepancyMonitor
  module Tester
    class Literals < Test
      include MetricWithLimit
```

```ruby

    def self.type
      :source
    end

    def name
      :literals
    end

    def run
      add_large_diff_note('number of literals', :danger) if diff > limit
      # if we go over these limits, given the avg. complexity of a method,
      # looking directly at the code is going to be more effective
      return if diff > 10 || limit > 20
      find_interesting_literals
    end

    protected

    def find_interesting_literals
      @set = {
        js: filter_uninteresting_literals(js_literals),
        csharp: filter_uninteresting_literals(cs_literals)
      }
      set_state(:table_var, order_by(:var))
      set_state(:table_value, order_by(:value))
      add_var_notes
      add_value_notes
    end

    def add_var_notes
      order_by(:var).each do |var, vals|
        missing_lang = false
        [:js, :csharp].each do |lang|
          unless vals.key? lang
            add_note('Variable ' + var + ' has no ' + lang.to_s + ' definition', :warning)
            missing_lang = true
          end
        end
        next if missing_lang || vals[:js] == vals[:csharp]
        add_note('Variable ' + var + '\'s assigments differ between languages', :warning)
      end
    end

    def add_value_notes
      order_by(:value).each do |val, vars|
        missing_lang = false
        [:js, :csharp].each do |lang|
          unless vars.key? lang
            add_note('Value ' + val.to_s + ' has no ' + lang.to_s + '
                definition', :warning)
            missing_lang = true
          end
        end
        next if missing_lang || vars[:js] == vars[:csharp]
        add_note('Literal ' + val.to_s + '\'s assigments differ between
             languages', :warning)
      end
    end

    def order_by(type)
      @table = {} if @table.nil?
      return @table[type] unless @table[type].nil? || @table[type].
         empty?
      table = {}
      @set.each do |lang, vars|
        vars.each do |item|
          next if type == :var && item[type] == :local
          to_id = item[type]
          to_add = item[type == :var ? :value : :var]
          # JS only does floats, so we need to unify for hashtables
          # to work like we want
          to_id = to_id.to_f if to_id.is_a? Numeric

          table[to_id] = {} unless table.key? to_id
          table[to_id][lang] = [] unless table[to_id].key? lang
          table[to_id][lang] << to_add unless table[to_id][lang].
             include?(to_add)
        end
      end
      @table[type] = table
    end

    def filter_uninteresting_literals(list)
      list.reject do |item|
        val = item[:value]
        # Long strings are mostly text instead of configuration
        next true if val.is_a?(String) && (val.length > 15 || val.strip
           .empty?)
        # n0, n1, nn...
        next true if val.is_a?(Numeric) && (val.zero? || val == 1)
        false
      end
    end

    def amounts
      {
        js: js_literals.length,
        csharp: cs_literals.length
      }
    end

    def limit_val
      0.1
      end

      def js_literals
        @jsl ||= find_literals([
          Parser::JavascriptStringLiteral,
          Parser::JavascriptDecLiteral
        ], [Parser::JavascriptAssignment], @data[:js])
      end

      def cs_literals
        @csl ||= find_literals([
          Parser::CSharpStringLiteral,
          Parser::CSharpNumLiteral,
          Parser::CSharpIntLiteral
        ], [
          Parser::CSharpAssignment,
          Parser::CSharpTypedAssignment
        ], @data[:csharp])
      end

      def find_literals(literal_types, ass_types, code)
        literal_types.map do |type|
          code.select_deep(type).map do |segment|
            var_name = :local
            ass = nil
            ass_types.each do |ass_type|
              ass = segment.find_parent(ass_type, 2)
              break unless ass.nil?
            end
            var_name = ass.var.text unless ass.nil?
            var_name = ass.var.statement.text if !ass.nil? && ass.var.
               respond_to?(:statement)
            {
              var: var_name,
              value: segment.value
            }
          end
        end.flatten
      end
    end
  end
end
```

Bibliography

[1] The LLVM Compiler Infrastructure projects built with llvm. `http://llvm.org/ProjectsWithLLVM/`. Accessed: 2015-07-12.

[2] Joel on Software things you should never do, part i. `http://www.joelonsoftware.com/articles/fog0000000069.html`, 2000.

[3] C Albert and L Brownsword. Evolutionary process for integrating COTS-based systems (EPIC): An overview. 2002.

[4] Andrew W Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, April 2015.

[5] Fred P Brooks Jr. *The Mythical Man-Month*, volume 10. ACM, June 1975.

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

[7] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. *Test generation through programming in UDITA*, volume 1. ACM, New York, New York, USA, May 2010.

[8] E Hajiyev. CodeQuest: Source Code Querying with Datalog. *Companion of the 20th Annual ACM SIGPLAN ...*, 2005.

[9] A R Hevner, S T March, J Park, and S Ram. Design Science in Information Systems Research. *MIS quarterly*, 2004.

[10] A Hunt and D Thomas. *The Pragmatic Programmer.* Addison Wesley, 2000.

[11] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. *Acm sigplan notices*, 41(1):141–152, January 2006.

[12] X Leroy. The CompCert C Verified Compiler. 2012.

[13] H Lötzbeyer, A Pretschner, and E Pretschner. Testing concurrent reactive systems with constraint logic programming. 2000.

[14] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[15] T J McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric, 1982.

[16] C Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution†. *Software Testing*, 2001.

[17] Fredy Navarrete, Pere Botella, and Xavier Franch. Reconciling Agility and Discipline in COTS Selection Processes. *2007 Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07)*, pages 103–113, 2007.

[18] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann Publishers Inc, April 1993.

[19] A M Pitts. Operationally-based theories of program equivalence. *Semantics and Logics of Computation*, 1997.

[20] A Seesing and H G Gross. A genetic programming approach to automated test generation for object-oriented software. *ITSSA*, 2006.

[21] G Sussman, H Abelson, and J Sussman. *Structure and interpretation of computer programs.* 1983.