Dynamic updates of mobile apps using JavaScript

by

Simon Spetz-Nyström

LIU-IDA/LITH-EX-A--15/021--SE

2015-06-04
Final Thesis

Dynamic updates of mobile apps using JavaScript

by

Simon Spetz-Nyström

LIU-IDA/LITH-EX-A--15/021--SE

2015-06-04

Supervisors: Mattias Wingstedt
VISIARC AB

Ola Leifler
IDA, Linköping University

Examiner: Kristian Sandahl
IDA, Linköping University
Updates are a natural part of the life cycle of an application. The traditional way of updating an application by stopping it, replacing it with the new version and restarting it is lacking in many ways. There have been previous research in the field of dynamic software updates (DSU) that attempt to salvage this problem by updating the app while running. Most of the previous research have focused on static languages like C and Java, research with dynamic languages have been lacking.

This thesis takes advantage of the dynamic features of JavaScript in order to allow for dynamic updates of applications for mobile devices. The solution is implemented and used to answer questions about how correctness can be ensured and what state transfer needs to be manually written by a programmer. The conclusion is that most failures that occur as the result of an update and is in need of a manually written state transfer can be put into one of three categories. To verify correctness of an update tests for these types of failures should be performed.
I would like to thank Visiarc for the opportunity to perform an interesting thesis project at their company. I would especially like to thank Mattias Wingstedt, who have been my supervisor at the company, for answering all my questions and helping me get started on the thesis.

I would also like to thank my supervisor at IDA, Ola Leifler, for continuously providing feedback and helping me with the direction of the thesis.
# Table of Content

## Chapter 1 Introduction

1. Background ........................................................................................................... 1
2. Visiarc ................................................................. 1
3.1 JavaScript ......................................................................................................... 2
3.2 Problem description......................................................................................... 2
3.3 Goal .................................................................................................................. 3
3.4 Research questions ......................................................................................... 3
3.5 Approach .......................................................................................................... 3
3.6 Scope and limitations ..................................................................................... 3

## Chapter 2 Related work .................................................................................. 5

1. Correctness of update ......................................................................................... 5
2. Time of change .................................................................................................. 6
3. State transfer ..................................................................................................... 7

## Chapter 3 Method ............................................................................................. 9

1. Update download ............................................................................................... 9
2. Update script .................................................................................................... 10
3.1 Change classification ...................................................................................... 10
3.2 Generation of update script .......................................................................... 11
3.2.3 Update points ............................................................................................. 13
3.3 Validation of update script .......................................................................... 13
3.4 Research method ........................................................................................... 14

## Chapter 4 Test application .............................................................................. 15

1. Application description ................................................................................... 15
2. Versions of the application ........................................................................... 16

## Chapter 5 Results ............................................................................................... 18

1. Uninformed and informed tests ...................................................................... 18
2. Types of failures ............................................................................................... 19
3.1 Non-configurable properties ....................................................................... 20
3.2 Initialization work .......................................................................................... 21
3.2.3 Change in storage structure ................................................................. 21

## Chapter 6 Discussion ....................................................................................... 23

1. Results ............................................................................................................. 23
2. Validation of update ......................................................................................... 23
3. Types of failures ............................................................................................... 24

## Chapter 7 Conclusions and future work ......................................................... 28

1. Answers to research questions ....................................................................... 28
2. Future work ...................................................................................................... 29

## References ........................................................................................................ 30
Chapter 1

Introduction

Updates are a natural part in the lifetime of all software. An update can be used to fix bugs in the original software, improve performance or even introduce entirely new features. In a mobile environment software updates are perhaps even more important due to the frequent changes in the mobile landscape.

The classical approach to updates is to stop the application, exchange it with the new version and then start the application again. This might be acceptable in some scenarios, but in other scenarios it would be much better to apply the update without the need to restart the application. For mobile applications it may seem like an unnecessary complication with dynamic updates, but apart from the convenience it can also be important, e.g. for applications running in kiosk mode.

Research in applying updates while an application is running, which is called dynamic software updates (DSU) has been extensive. However, it has been more limited for both applications using such dynamic programming languages as JavaScript and applications running in a mobile environment. Both of these subjects are dealt with in this thesis.

1.1 Background

1.1.1 Visiarc

Visiarc is a company located in the center of Linköping, Sweden. They focus on making multi-platform mobile apps using their framework Coffee (Code Once Framework For Experiences Everywhere) and visual tool Coffeemaker. With these tools an environment to run JavaScript is included, which allows for writing the app logic in one piece of JavaScript code that can be used on both iOS and Android.

Visiarc uses these tools in the development of their apps, which are both client commissioned and their own in-house productions.

The JavaScript code is considered part of the content of the application by the mobile operating system, and not part of the code. This means that more advanced solutions for updating of the app can be allowed than the traditional way that the operating system supports. While the solution described in this thesis can be useful in the apps deployed by Visiarc the main focus has been to improve the development process. By allowing dynamic updates Visiarc will be capable of quickly testing small code improvements without the need to restart the application.

---

1 Kiosk mode means that the application is the only thing accessible on the device
1.1.2 JavaScript

JavaScript is a scripting language originally included in Netscape’s Navigator 2.0 browser. It was later standardized in a language called ECMAScript. In this report the term JavaScript is used even if ECMAScript might in some ways be more technically correct. This was chosen because JavaScript is a more well-known term and it is often used as a synonym for ECMAScript.

JavaScript was originally designed for the web, but the language as described in the standard does not assume any specific host environment. In the context of this thesis the JavaScript code does not actually run on the web. Instead a custom host environment using the scripting engine SpiderMonkey is used.

The syntax of JavaScript intentionally resembles the syntax of Java. However, some features of the language differ substantially from how Java works. The most significant differences for this thesis is the prototype-based inheritance and the dynamic properties. These differences are briefly described below.

1.1.2.1 Prototype-based inheritance

Unlike Java and other similar languages, JavaScript does not use classes. Instead objects can be created using object literals. To support inheritance objects can also be created using constructors. A constructor has a property called `prototype` that is used for the prototype-based inheritance. When an object is created with a constructor in a `new` expression the object will receive an implicit reference to the `prototype` property of the constructor. When a property of the object is accessed the object itself is first checked for the property and then the prototype. A prototype can also contain a reference to another prototype, forming a prototype chain that is traversed in order to find a property.

1.1.2.2 Dynamic properties

In class-based object languages all properties of an object are defined in the class of the object. This is performed statically and there are in general no ways of adding or deleting properties during runtime. In JavaScript on the other hand, properties can dynamically be added simply by assigning a value to a property that did not previously exist. A property can also be removed using the `delete` keyword.

JavaScript also uses dynamic typing, which allows for changing the type of a property simply by assigning a new value to it.

1.2 Problem description

The overall problem this thesis handles can be described as applying updates to mobile applications in an efficient and silent way. This problem can be divided into two subproblems, getting the update to the device and applying the update. This thesis focuses on the second part of the problem, but the first part is also described briefly.

When getting an update to a mobile device it is important to reduce the download size. Downloads are both slow and possibly expensive for mobile devices so downloading should be limited to what is necessary. This thesis does not only deal with the deployment scenario where versions are well-defined and can be easily compared server-side but also with the development scenario where versions are not well defined. Therefore the problem handled is how to determine what has changed without full knowledge of what is available on the client.

The update process shall be as nonintrusive as possible, allowing the user to use the application without even noticing that the update occurs. The only way the user should be able to notice the update has even been performed is by the new functionality it provided. It must
also be persistent, meaning that if the application is restarted the new version will be the version running.

1.3 Goal

For the problem of getting the update to the device there is only one goal for this thesis: download only the files that have changed.

The dynamic update part of the problem has more goals. In the previous research on the subject of dynamic updates there have been different goals depending on the application. The common factor is that the update shall be able to be performed without the need to stop the application, which is called continuity in the list of goals below. The other goals are based on this specific application of dynamic updates.

- Continuity. The update shall be performed during runtime with an interruption time of the execution as short as possible.
- Programmer transparency. The programmers of the mobile applications should not have to change their development process to allow for the updates.
- Flexibility. As many changes as possible shall be possible. In the optimal case all updates that are possible in the traditional way shall be possible.
- Correctness. After the update has been applied, the application shall act in the same way as if the updated version had run from the beginning.

1.4 Research questions

The thesis deals with the following research questions:

1. What kind of information about the change is needed to verify correctness of an update?
2. How much of the state transfer can be automatic and how much is required to be specified by the programmer?

1.5 Approach

The problem is approached by looking at previous work in the field of dynamic software updates to determine what ideas and techniques can be applied to the domain of this thesis. A solution is then designed and implemented based on this. The resulting solution is evaluated according to the goals listed in 1.3 and experiments of using the implementation are used to answer the research questions.

1.6 Scope and limitations

There are many interesting topics to cover when it comes to dynamic software updates. This thesis limits itself to examine the questions listed in 1.4 and the goals listed in 1.3. Another very important topic for dynamic updates, especially when performed like in this thesis, is security. To perform the update the application executes code from a remote source (the server) which means security is very important. However, the security questions are not dealt with in this thesis.

The update process described and implemented is not a general solution for all JavaScript applications, it depends in some parts on the execution environment for the applications developed at Visiarc. The thesis contains some discussion on how far the results and method
used can be generalized to different JavaScript applications and environments, but no attempt was made to implement a completely general solution.

This thesis aims to answer questions about what is necessary to be able to perform dynamic updates. No ethical or societal aspects could be connected to this topic and therefore no such issues are discussed.
Chapter 2

Related work

This chapter presents some of the work that has previously been produced in the field of DSU. There have been a large amount of approaches to the subject. For example, Seifzadeh et al. compare 48 different approaches in their survey of the topic [18]. This chapter will focus on some of the major concepts and techniques used.

2.1 Correctness of update

One of the most important aspects to consider when performing dynamic updates is to ensure that they are correct. However, there exists no generally agreed upon definition of what is meant by correctness of an update. Depending on the definition used previous work have presented different ways of ensuring that an update is correct.

In a paper by Bloom and Day [5] the correctness condition of a replacement is expressed as “continuation abstractions are preserved or invisibly extended by a replacement”. They note that it is not enough for the new instance to fulfill the original abstraction, which is the reason for introducing continuation abstractions. An example of a generator for unique identifiers (UID) is given. If the specification of the UID generator says that it will produce 1000 different identifiers before repeating itself an implementation that counts from 10 to 1010 cannot be replaced by an implementation that counts down from 1000 to 1. Both implementations fulfill the original abstraction, but the replacement does not fulfill the continuation abstraction. If the first implementation has produced 10, 11 and 12 at the time of replacement the continuation abstraction will be that the new implementation must produce 997 distinct values that are not 10, 11 or 12 after which the original abstraction have to be fulfilled. A more formal definition is given in the PhD thesis by Bloom [4], which allows for formal proofs of correctness. However, such a proof requires that the abstractions are well defined.

Gupta et al. [6] proposed the use of reachability for the definition of validity. They define validity of an update with a guarantee that the process will reach a reachable state of the new program within a finite amount of time. They do not have any restrictions on the state reached other than that it must be reachable in the new program. However, they do acknowledge that for the state transition function to be useful the reachable should in some way correspond to the state that would have been reached if the new program had been running from the beginning and received the same input as the old program had. For this definition of validity it is proven that it is undecidable to check validity of an update given the two versions, the current state and the state transformation used. However, Gupta et al. provide sufficient conditions for states where the validity of an update can be formally proven.

Even though the use of reachability as basis for validity is good in the sense that validity can be formally proven there are other properties of the definition that makes it questionable. Hayden et al. [7] argue that the definition is both too permissive and too restrictive. To show this they use the example of version 1.1.2 of vsftpd, which introduced a feature to limit the amount of connections from a host. The expected behavior would be to keep all connections alive during the update, but if the number of connections is higher than the limit and these connections are never closed the reachability property is violated. However, using reachability as definition of validity would allow an update to simply terminate all active connections, which
violates the fundamental goals of a dynamic update. They go on to argue that it is unreasonable
to define validity in a general way and propose that programmers specify the expected behavior
in what is called client-oriented specifications (CO-specs). Using these CO-specs allowed them
to verify the update using off-the-shelf verification tools. They used the symbolic evaluator
Otter [16,18] and verification tool Thor [14] for the verification. Both tools gave some valuable
information. However, Thor could only fully verify some of the updates. Otter is faster and
could check all CO-specs but does not perform full verification. The authors argue that because
their approach does not depend on the verification tool used they will benefit from future
improvements of verification tools without the need to change how CO-specs work.

In another article by Hayden et al. [9] testing is used to determine whether an update is
correct or not. A test suite containing the intersection of tests for the old and the new version is
used and the application is updated during test execution. Because the tests shall pass both for
the old version and the new version it should not matter when during execution the application
is updated. This approach has the benefit that it ensures the behavior of the application is as
expected, but one major drawback is that update validity cannot be proven in advance. For
every update the tests have to run to ensure the update is correct. The correctness also depends
on the tests, with bad or insufficient tests correctness cannot be guaranteed. Hayden et al. do
not reflect on exactly what is needed for the tests to be good, but simply acknowledge that tests
are not a complete measure of correctness but generally cover the most important parts of the
application.

2.2 Time of change

Even though the dynamic updates are performed when the application is running it is not
possible to apply the update at an arbitrary time, it has to be performed at some kind of safe
state. It has been proven that it is undecidable to check validity (expressed as reachability) of
an update given the two versions, the current state and the state transformation used [6].
Therefore it is not possible to determine properties that are both sufficient and necessary for an
update to be correct.

One way to decide what is a safe state was suggested by Kramer et al. [13] with the
introduction of the concept of quiescent states, They use a model of the software as nodes, that
interact with each other through transactions. A node is defined to be quiescent if the following
four properties are fulfilled:

1. it is not currently engaged in a transaction that it initiated
2. it will not initiate new transactions
3. it is not currently engaged in servicing a transaction
4. no transactions have been or will be initiated by other nodes which require service from
the node

This concept have been considered to be too strict in the sense that it causes high disruption.
Therefore Vandewoude et al. [23] introduced the concept of tranquility. A node is defined as
being in a tranquil state if it has these four properties:

1. it is not currently engaged in a transaction that it initiated
2. it will not initiate new transactions
3. it is not actively processing a request
4. none of its adjacent nodes are engaged in a transaction in which it has both already
participated and might still participate in the future

The difference in the third and fourth properties for tranquility and quiescence come from
two observations Vandewoude et al. have done. The first is that a node can be replaced even if

---

2 Vandewoude et al. use the term status instead, but here the term state is chosen to be consistent with the
original definition of quiescence.
a transaction it is involved in is active. The only requirement is that the node will not be used more or that the node has not been used yet.

The second observation by Vandewoude et al. was that a black-box design implies that the initiator of a transaction does not know of, and therefore does not depend on, the nodes that are involved in subtransactions created by nodes in the transaction. The example given is a transaction with three participants: X, Y and Z. X is the initiator of the transaction and first communicates with Y and then with Z. Before Z responds to X it starts a (sub)transaction with Y. Tranquility results in the possibility to update Y at the point where communication with Y have ended and communication with Z have not begun. This is possible because from the point of view of X, Y will not be used anymore in the transaction. The result is therefore consistent even if X and Z use different Y within the same transaction.

The main disadvantage of tranquility as opposed to quiescence is that tranquility is not guaranteed to be reached in bounded time. Tranquility is something that occurs naturally whereas quiescence is something that is actively achieved by passivating nodes. A system using tranquility would therefore need to have a fallback mechanism where quiescence can be used if updates shall be guaranteed.

The definitions of quiescence and tranquility are both written for distributed systems, but the same ideas can be applied to non-distributed systems as well. In a classic procedural system the procedures can be considered the nodes and procedure calls being the transactions. A similar (though not equivalent) definition of quiescence would then be that the changed procedures are not on the activation stack. This definition of a safe time to update is used by several DSU systems, including Ksplice [2], Jvolve [21] and OPUS [1].

A property somewhat similar to tranquility for procedural programs is con-freeness, which was introduced by Stoyle et al. [20]. A point in the program is considered con-free for an update if changed types will not be used by the active expression after the update. This allows for changing of active methods, as long as they are con-free.

Hayden et al. [9] have performed tests of the safety of updates using the stack criterion, con-freeness and programmer specified update points in the beginning or end of a long running loop. Their results was that neither con-freeness or the activation stack criterions guaranteed correctness. For con-freeness there was slightly less than 1% of the possible update points that failed. The corresponding number for the stack criterion was less than 0.02% and the problems only occurred for one of the three programs that was tested. Though these numbers are a large improvement to allowing updates at arbitrary times (which failed for approximately 13% of the update points) it is still possible for failures in the tests to occur. For the manually inserted update points at the beginning or end of long-running loops no failures were encountered in their tests. The main downside of the manual approach they found was that updates were possible at much fewer places, meaning the wait time until an update occurs will be longer. However, in the programs tested the longest time for a loop iteration was less than 10ms, a time the authors argue is small enough to not be significant.

### 2.3 State transfer

When performing a dynamic update the new version of the code should start execution from a state that is in some way equivalent to the state where the old version of the code stopped executing. This problem is very hard in its most general case. Therefore many of the previous approaches leave it to the programmer to provide a mapping from states in the old program to states in the new program. Gupta et al. [6] argue that it is necessary for the developer to provide the state mapping because of the semantic knowledge only the developer has. They use the example of renaming a variable from x to y as something that can only be understood by the developer. Another example that shows the difficulty in automatic transfer of state is if the implementation of an abstract data type changes. If a singly linked list is used in the old program
and a doubly linked list is used in the new program all back links have to be added. How this process is performed is nothing that can be automatically decided.

However, in some cases state transfer without help from the programmer is possible and in some cases even simple. For stateless entities a state transfer is obviously not necessary. If none of the state variables have changed name and/or semantics the old values can simply be copied to the new version of the code. In this case it is of course hard to know if the semantics have changed or not. Kitsune, a framework for dynamic updates, [8] uses this reasoning to allow for automatic migration of some variables. By default, Kitsune considers all global variables to contain state information and therefore performs migration for all global variables. Local variables are not considered to contain state information. If nothing else is specified the old values will simply be copied to the new program. For new variables or variables with changed type the transformation generator of Kitsune will fail if the programmer have not specified a transformation.
Chapter 3

Method

In this chapter the methods for performing and evaluating the dynamic updates are described. The update process can be described in three parts: determining and downloading the changed files, creating the update script and applying the update script. The first two parts are performed mainly on the server and in a separate thread in the application, meaning it results in no interruption for the user. Applying the update script however is performed synchronously and blocks any other activity from the app.

3.1 Update download

Although an update has to be performed at a specific point in time to not break consistency the download of the update can be performed at any point in time. Therefore the download process is performed in a separate thread, continuously downloading new files. The client and server are connected by a TCP socket and the file transfer occurs according to the following protocol:

1. Server detects that files have changed and sends a list of file names and checksums to all clients. (FILES_CHANGED)
2. Client compares the checksums to its own checksums
   a. Send a list with the files (of all types) that have changed to the server (GET_FILES)
   b. For all JavaScript files that have changed, send the entire file and its filename to the server (OLD_FILE). When all old files have been sent a message is sent informing the server that file transfers are complete (OLD_FILES_END).
3. The server responds to the messages according to this:
   a. GET_FILES: For each file in the list, send the latest file to the client (NEW_FILE)
   b. OLD_FILES_END: Generate an update script and send it to the client (UPDATE_SCRIPT)

In a deployment scenario where the versions are well-defined, the first two steps are not needed and can be replaced by the client simply sending its version number. The update script generation can potentially be moved to the client, which would mean there is no need for the OLD_FILE and UPDATE_SCRIPT messages. However, it was chosen to perform this on the server instead because of two reasons.

1. It allows for custom update scripts written by a developer. This can be necessary if the update script generation cannot handle an update. It can also be used to create "update scripts" that are not update scripts, but that are used for debugging, e.g. by inspecting the value of a variable.
2. Less burden on the client.

When the client sends a GET_FILES message the file list is saved and as soon as the first NEW_FILE message is received a database transaction to insert the files into the database transaction is started. The transaction does not commit until all files have been received, ensuring atomicity of the process. Either all files will be updated or none of them will.
The files are reloaded from the database when the app starts, but in order to make the app updated without restarting it, the update script is necessary.

### 3.2 Update script

JavaScript is a dynamic language, which results in some complications for dynamic updates in particular in the difficulty of verification and testing [3]. However, it also results in some advantages because properties can dynamically be added, changed or removed using the language itself. All properties of objects created with a constructor can be changed by changing the properties of the prototype, thus allowing changes in inherited properties as well.

In fact, simply executing the entire changed file is sufficient to provide some of the features needed in a DSU system. Functions, objects and properties defined in the file will be changed to the latest version. New functions will also be added. However, this approach of course has its limitations as well. For example, it can’t deal with deletions of properties, the old version will still be used. It might also be dangerous to overwrite properties, because a property holding state may be overwritten. For these reasons the files first have to be analyzed to determine what changes have occurred. With this knowledge it is possible to generate more safe update scripts.

#### 3.2.1 Change classification

In order to determine what parts of the new code is changed code, added code and what code has been removed the Simple_Tree_Matching_Algorithm by Yang [24] is used to match AST (Abstract Syntax Tree) nodes from the old code to nodes of the new code. This algorithm is described in Figure 3-1 below. The values of the nodes mentioned in the algorithm refers to both type (e.g. if) and primitive values (e.g. name of symbol).

```plaintext
Simple Tree Matching(A,B)
if A has different values than B then return 0
m := A.children.length
n := B.children.length
M[i, 0] := 0 for i = 0,...,m
M[0, j] := 0 for j = 0,...,n
for i from 1 to m
  for j from 1 to n
    w := Simple_Tree_Matching(A.children[i], B.children[j])
    M[i, j] := max(M[i, j-1], M[i-1, j], M[i-1, j-1] + w)
return M[m, n] + 1
```

Figure 3-1. The Simple_Tree_Matching algorithm

Nodes in the old code are mapped to nodes in the new code with an algorithm similar to the algorithm to find the string correction in the original algorithm for the string-to-string correction problem [22] that Simple_Tree_Matching is based on. The algorithm, which I have chosen to call Node_Mapping, is presented in Figure 3-2.
Node Mapping (A, B, M)

mappings := empty set
i := M.columns
j := M.rows
while i > 0 and j > 0
    tmp := max(M[i-1, j], M[i, j-1], M[i-1, j-1])
    if tmp != M[i][j]
        mappings := mappings ∪ (A.children[i] -> B.children[j])
        i := i - 1
        j := j - 1
    else if tmp = M[i-1][j]
        i := i - 1
    else
        j := j - 1
return mappings

Figure 3-2. The algorithm used to map new nodes to old nodes

A node that exists in the mappings set is either changed (if the code of the nodes differ) or unchanged (if the code of the nodes are equal). A node from the old/new code that is not available in any mapping is considered removed/added.

This classification can be performed at several levels, corresponding to the amount of recursive calls in the Simple Tree Matching algorithm. How many levels shall be analyzed is chosen by balancing the amount of gained information with the increased time of analyzing more levels. For generation of the update script, only the top level is analyzed. For the tests more detailed information is necessary. A couple of experiments were performed to decide how to present the information for the tests. The results of these tests can be found in 5.1.

3.2.2 Generation of update script

The generation of the update script is relatively simple in most cases. The update script is based on the original file but removes all code blocks that have not changed. This is done to perform as few unnecessary and potentially harmful operations as possible. A piece of code can for example be supposed to run at the start of the application to initialize some variable to a value corresponding to the start state. If this initialization code has not changed the safest thing to do is to not run it again. If the initialization code has changed it is harder to tell what should be done. It can be considered likely that the variable is now used in a different way, which means that the current value is invalid in the new version. On the other hand, running the changed code will give a value that is valid, but might not be what is expected in the current state. The reasoning behind making all changed code part of the update script is that it gives values that are valid in the new version. In the case where this is not enough the developer will have to manually write the appropriate state transition function.

For the removed code, the update script generator only considers the case of assignments. If an assignment of a property has been removed, this probably means that the property shall no longer exist in the new version and is therefore deleted with the delete keyword.

This simple approach is well suited for most situations, but there are a few cases where special consideration is taken depending on the type of change. These situations are described in the headings below.

3.2.2.1 Unchanged property assignment on changed variable

One case where not running the unchanged code can actually be harmful is when the unchanged code contains the creation of a property on a changed variable. An example can be seen in the two code snippets in Figure 3-3.
Dynamic updates of mobile apps using JavaScript

### 3.2.2.2 Classical inheritance

JavaScript uses prototypal inheritance instead of classical inheritance. Due to the large amount of developers with background using classical inheritance there exist several attempts at mimicking classical inheritance in JavaScript. This thesis will handle the technique called Simple JavaScript Inheritance, suggested by John Resig [17], but the approach is general enough to handle similar problems that might occur with other implementations of JavaScript equivalences of classes and traditional inheritance.

Like many of the features of JavaScript, the dynamic nature of its prototypal inheritance is beneficial for dynamic updates. A change in an object that is used as a prototype will result in changes for the objects that use the prototype as well. Unfortunately when a class is redefined in Simple JavaScript Inheritance, it does not have the requested effect for existing instances or subclasses. Instead of changing an existing prototype a new prototype is created when the class is defined. This works well in the regular scenario, but not for supporting DSU. When the instances (or subclasses) were created the current prototype of the class was set as the prototype. If a new prototype is created everything that point to the old prototype will not be changed.

To support updates of superclasses class definitions are handled separately by the update script generator. They are handled by assigning the return value of the `subClass` function to a temporary variable. The properties of the prototype for the temporary object are then copied to the properties of the prototype of the variable used for the class.

### 3.2.2.3 Object.defineProperties

There are multiple ways to define properties on an object in JavaScript. Perhaps the most common way is with the dot or brackets syntax (i.e. `o.newProp = "value"` or `o["newProp"] = "value"`). It is also possible to use the function `Object.defineProperties` (or `Object.defineProperty`) that was introduced in ECMAScript 5.1.

In most cases the update script generator only looks at changes in top level statements, meaning that if arguments to a function call has changed no consideration is taken as to how they have changed. Due to the fact that the default behavior of `Object.defineProperties` is to make the properties non-configurable and non-writable it is important to look at the arguments to allow for more types of updates. Therefore the change classification is performed on the second argument to `Object.defineProperties`. The second argument is an object with the properties to define. By running the change classification on the object all properties that does not have a changed definition can be removed from the update script.

```
var o = {a:1};
o.b = 11;

var o = {aa:1};
o.b = 11
```

**Figure 3.3.** Example of unchanged property `b` on changed variable `o`
Object.defineProperties is also handled in a special way in order to support the unchanged properties mentioned in 3.2.2.1.

### 3.2.3 Update points

Previous research (see 2.2) have shown that the most appropriate way to place update points is to do it manually, using a simple design pattern. Therefore this approach has been chosen in this thesis as well.

The design pattern suggested in previous research is for the update points to be in the beginning or end of long-running loops in the program. In JavaScript applications in general, and the applications tested in this thesis in particular, the long running loop is the event loop. This loop is part of the execution environment and not of the application, which makes it harder to change directly. However, a similar effect can be achieved with the `setInterval()` method if the HTML DOM is available, and a corresponding method if the application runs in another execution environment. Placing an update point in a call to such a method means the update will be attempted with a regular interval. Due to JavaScript being single-threaded, this will occur between any event handlers and not in the middle of an event handler, effectively placing it in “the beginning” of the event loop.

In this thesis, the applications that are updated are graphical. This means that it might be inappropriate to perform an update at a time that would lead to a large unexpected graphical change. Although the application may be considered inactive in the way that no events are handled, the user experience can suffer. If the application is suddenly changed when the user is in the middle of something the user can be confused and annoyed. The extreme example would be the removal of a feature just as the user is about to use the feature. This reasoning leads to the approach used in this thesis, which is to place an update point at transitions between different “screens”. This is a good place to update as the large change in how the application looks gives a natural change of context. The biggest potential problem of this approach is of course that these changes might not occur very often. Such transitions are often the result of the user actively doing something to change view and how often this happens is therefore dependent on the user.

Furthermore, many applications are not balanced in the expected time for different screens. One example is games, that often have a few menu screens and one main game screen. The expected time spent in the game screen far outweighs the time spent in the menu screens.

### 3.3 Validation of update

The approach chosen for validating updates is based on testing. Hayden et al. [9] showed that testing can detect problems with updates that are considered safe using other definitions. While the same kinds of problems could also be detected with CO-specs, the approach of testing is better for programmer transparency, because tests used to verify correctness of the application can be reused for verifying correctness of an update.

One thing Hayden et al. did not deal with in their paper was what kind of tests are necessary to be sure that the update was indeed correct. In the paper they acknowledge that testing is an incomplete measure of correctness, but argue that they typically cover the most important program behaviors.

For this thesis, the classification of changes described in 3.2.1 is used to determine if knowledge about the changes provides tests that in a better way validates the updates. There are several reasons why this knowledge could provide for a better validation. In usual tests, no updates are considered, which might mean that update-specific types of failures will not be found. For example, non-existence of features is generally not tested but if a feature is removed in an update such a test might be necessary.
For the application used in the experiments, Visiarc uses exploratory testing and this method of testing was also used for validating the correctness of the updates. Unlike traditional testing with predesigned test cases exploratory testing performs learning, test design and test execution simultaneously. This approach to testing have been shown to find a similar amount of defects as traditional testing with test cases, but provides fewer false defect reports [11] [10]. Due to exploratory testing being more free than the traditional test based versions, the success of exploratory tests depend more on the person performing the tests. Research have been performed examining both the importance of the personality and knowledge of the tester. Shoab et al. [19] concluded that having an extrovert personality type and a high IQ level are factors that give a good exploratory tester and Itkonen et al. [12] suggested that exploratory testers apply their knowledge of the domain during testing, which improves the results.

Tests were performed after an update had been applied. A test was considered to fail if the behavior of the updated application was not the same as the behavior of the same version installed without update. Because the application under test is in the development phase this means that all versions have some expected bugs. If these bugs are not visible in the updated application it is considered a failure of the test. Correct behavior was defined as the behavior when the application was installed on a phone with no previous installation of the application.

In order to see whether the change classifications provided valuable guidance or not two different types of tests were performed. First the updated application was tested in a way similar to how the application would have been tested if installed in the classical way and then knowledge of the actual changes were inserted into the testing.

I performed all the tests myself so in order to perform tests both with and without detailed knowledge of the change the tests were first performed without the extra knowledge. After that I looked at the change analysis and determined whether any more tests could be considered necessary.

### 3.4 Research method

To find the answers to the research questions experiments were performed on the update process described above. To get a realistic scenario of updates that can occur an application that had been previously developed at Visiarc was used for the updates. This application is further described in Chapter 4.

Tests were performed according to what is described in 3.3. The test failures were analyzed further to determine the reasons behind them. These reasons were then used to answer the research questions.
Chapter 4

Test application

For evaluation of the methods described in Chapter 3 different versions of an application developed by Visiarc was used. The application itself is described in 4.1 and the versions tested are described in 4.2. An image of the game in version 1 and 7 can be seen in Figure 4-1 below.

4.1 Application description

The application used is an unreleased game called Ordsmart. The gameplay is similar to the Hangman game. The goal of the game is to guess a word with a limited amount of guesses (the amount varies between the versions). When the user guesses a word information is given for all letters with one of the following indications:
1. The letter is not part of the word
2. The letter is part of the word, but in the wrong position
3. The letter is part of the word and is in the right position

A score is awarded to the player based on how many guesses was required before the correct word was guessed and the game is over if the user fails to guess the correct word with the allowed amount of guesses.

The game features two modes of gameplay: solitary and co-op. In the solitary mode the player performs all the guesses and in the co-op mode the two players take turns guessing.

In later versions of the game power-ups were introduced. They can be bought in a store and the effects include removing letters from the keyboard and granting extra wishes.

### 4.2 Versions of the application

For the purpose of this thesis different versions of the application were downloaded from the Git history and updates were performed between the versions. The versions have been numbered 0-7 and cover a time period from September 2014 to March 2015. A summary of the main changes in a version can be seen in Table 4-1 below. The changes listed are those that are the most visible to the user, and not the changes at a code level that can be seen with the change classification described in 3.2.1. Of course, all newer versions also contain bug fixes and the table is in no way a complete listing of all changes. The table is provided simply as a way of demonstrating the typical changes encountered.
Table 4-1. Description of major changes between versions

<table>
<thead>
<tr>
<th>Version number</th>
<th>Main changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>Major design changes. Introduction of more screens.</td>
</tr>
<tr>
<td>2</td>
<td>Added replay of guesses.</td>
</tr>
<tr>
<td>3</td>
<td>Added power-ups.</td>
</tr>
</tbody>
</table>
| 4              | Added store to purchase power-ups and extra guess.  
Unintentionally introduced bug for co-op. |
| 5              | Login/User creation required at startup. Changed visuals for power-ups. |
| 6              | Allowed guesses decreased. |
| 7              | Various graphical changes. |

During the testing the latest versions of the various libraries needed for the application were used. Only the actual code part of the application was updated. This means that bugs might have been introduced in the older versions, if they depended on some old behavior in a library. The behavior of the different versions is not necessarily the same as the behavior was when a version was originally created. In the tests the correct behavior is determined by the behavior with the latest libraries, even if this means more bugs.
Chapter 5

Results

In this chapter the results of the tests are presented. In 5.1 the amount of failures found when testing with and without guidance about which changes occurred. In 5.2 the failures encountered during all of the tests are further described.

5.1 Uninformed and informed tests

In Table 5-1 below the amount of failures encountered during testing are listed. The type of failure described in 5.2.1 happens very early in the update process and prevents any further testing. Therefore the simple pattern described to create a manual update script was used in these cases to allow proper testing. The numbers for failures found with this fix are written in italics.

All updates tested were for two consecutive versions (i.e. from n to n+1). The tests were performed completely independently of each other, which means that the tests from e.g. version 1 to version 2 started at a fresh install of version 1, and not with an application that had been updated from version 0 to version 1.

Table 5-1. Test results

<table>
<thead>
<tr>
<th>Update</th>
<th>Update points</th>
<th>Failures found (uninformed)</th>
<th>Additional tests with more information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0→1</td>
<td>2</td>
<td>2</td>
<td>5 (0 failed)</td>
</tr>
<tr>
<td>1→2</td>
<td>5</td>
<td>1+0</td>
<td>1 (0 failed)</td>
</tr>
<tr>
<td>2→3</td>
<td>6</td>
<td>1</td>
<td>1 (0 failed)</td>
</tr>
<tr>
<td>3→4</td>
<td>5</td>
<td>1+6</td>
<td>2 (0 failed)</td>
</tr>
<tr>
<td>4→5</td>
<td>4</td>
<td>1+3</td>
<td>3 (0 failed)</td>
</tr>
<tr>
<td>5→6</td>
<td>4</td>
<td>1+1</td>
<td>1 (0 failed)</td>
</tr>
<tr>
<td>6→7</td>
<td>4</td>
<td>0</td>
<td>1 (0 failed)</td>
</tr>
<tr>
<td>Total</td>
<td>30</td>
<td>7+10</td>
<td>14 (0 failed)</td>
</tr>
</tbody>
</table>

Not all possible update points were tested. Instead a subset of the possible update points were selected. The selection process used an exploratory approach, in order to fit with the approach to the tests performed. Because the tests were performed one update at a time, the selection of update points for later tests were partly based on results from earlier tests.

Failures that occur for several update points, but which are clearly a result of the same faults in the update process, are only presented as one failure. However, failures with the same reason but different behavior can be presented as several failures. For example, three of the failures in the update from version three to four have the same exact reason, but appears in different ways depending on when the update script is applied.

For the informed tests, the first decision that had to be made was what kind of information shall be provided. The information shall be as condense as possible, while still giving enough information to be helpful. As a first attempt the same information that is used for the generation of the update script was provided. This means information about which nodes at the top level have changed, been added and been removed. To give some added context the code for the old
and/or new node was presented. Due to this resulting in a lot of code in the output it was considered too much information to be easily accessible. An attempt was made to present just the beginning of the code for the nodes, but this provided too little information to be usable.

To show the relevant differences without presenting large code blocks experiments were made with presenting change information at all levels, with the current level indicated by indentation. With this solution it was relatively easy to determine what changes were relevant, but a lot of information that was not relevant was presented. The conclusion from this experiment was that the changes that were the most interesting were expressed at the property level. Unfortunately property changes were located at many different AST levels.

The best solution that was examined was to look at all levels of the AST, but only show changes to the AST nodes that represented a change in a property or the assignment to a property. The changes are indented in order to show what level the change is at. An example of output for a file can be seen in Figure 5-1. Most files have many more properties changed than this example.

| Change in assignment to cm.Backspace (OLD: cm.Backspace) (starts at line 9) |
| Added property onLongPressBegin (starts at line 134) |

**Figure 5-1.** Example output from change information

This solution of course examines a lot more nodes than if only the top level of the AST was examined. For an update from version 0 to 1 an average of 21,234 nodes are examined per changed file, as opposed to 16 if only the top level is used. This also results in approximately 20 times longer execution time.

Even though this method was the best, it was still not considered sufficient to decide what tests to perform. Without knowledge of exactly what changes have been performed within a function it was hard to design new tests. However, when more detailed information was presented it was near impossible to actually find the relevant information in all the text that was written to the output. A better way to display the data was needed, but implementation of such a solution is not part of this thesis. Therefore the difference tool Beyond Compare³ was used in order to see in detail which changes had occurred. This resulted in a total of 14 additional tests that were performed. All of these tests passed.

### 5.2 Types of failures

The failures encountered can be divided into three categories: non-configurable properties, initialization work and change in storage structure. The distribution of the different failures can be seen in Table 5-2 below. Both the total amount of failures in a category and the amount of updates where a failure of a type occurred (out of the seven tested) is presented.

| Table 5-2. Failures by category |
| Updates failure occurred for | Total failures |
| Non-configurable properties | 4 (57%) | 4 |
| Initialization work | 4 (57%) | 10 |
| Change in storage structure | 2 (29%) | 2 |

One failure did not fit into any category, and that was a failure that occurred in the update between version 5 and version 6. In version 6 the amount of guesses allowed was decreased from seven to six. This is similar to the change of vsftpd that Hayden et al. discussed where the feature to be able to limit the amount of clients was introduced. It can be hard to determine what

³ Beyond Compare is developed by Scooter Software and available at [http://www.scootersoftware.com/](http://www.scootersoftware.com/)
is the correct behavior in this scenario. Surely a user that started a game with seven available guesses expect to have seven guesses when the game is re-entered. On the other hand, this violates the rules of the new version. The behavior for the automatic update script was to allow seven guesses if and only if six guesses had already been made. This was not reported as a failure because it might be what is wanted. A user that leaves a game with one guess left do definitely not want to come back to the game and suddenly have no guesses left. The same could be said for a player with two or more guesses left, that might not want to come back to the game with one guess less left. In that scenario, it is at least still possible for the player to adjust the tactics to fit better with the new rules.

In the case where a seventh guess was allowed it was also allowed to use the power-up to receive an extra guess, making it a total of eight guesses as opposed to the seven guesses that are allowed with an extra guess in the new version. Again, this can be considered expected behavior. The problem arises when an eighth guess is attempted. When the limit of guesses was decreased the animation for adding the extra guess was changed to work with a total amount of seven guesses. The eighth guess causes the animation to look bad, and that is what was reported as a failure. Again, this failure does not fit into any of the categories above.

5.2.1 Non-configurable properties

One of the failures that occurred in the largest amount of updates was for non-configurable properties. This means that there have been a change in a property that is not configurable. When the automatically generated update script tries to change that property an exception is thrown. This type of failure can therefore be easily detected as soon as the update script runs. This kind of failure can potentially be impossible to solve at the current level of the update process. However, some versions of this failure can be circumvented using a manually created update script. All failures of this type that was encountered for the tested application was when a JavaScript specific implementation for lazy initialization of a singleton was used. The pattern can be seen in the code of Figure 5-2.

```javascript
var NAMESPACE = NAMESPACE || {}; (function() {
  function LazySingleton() {
    // constructor code
  }
  LazySingleton.prototype.method = function() { // code
  };
  Object.defineProperty(NAMESPACE, "LazySingleton", {
    get: function() {
      var sInstance = new LazySingleton();
      Object.defineProperty(NAMESPACE, "LazySingleton", {
        value: sInstance,
        configurable: false
      });
      return sInstance;
    },
    configurable: true
  });
}());
```

Figure 5-2. Lazy initialization of singleton in JavaScript

If a change occurred in one of the methods of the singleton the update script will re-run the entire code block within `(function() { ... })();`. If the `get` method has been used on the singleton an exception will be thrown because of an attempt to change a non-configurable
property. This can be avoided using a manual update script written according to the pattern described in Figure 5-3.

```javascript
Object.getPrototypeOf(NAMESPACE.LazySingleton).method = function() {
    // New code
}
```

Figure 5-3. Pattern to solve problems with non-configurable properties

Please note that the code in Figure 5-3 is safe whether the singleton have been initialized or not, because the `get` method is implicitly called when evaluating the argument to `Object.getPrototypeOf`.

### 5.2.2 Initialization work

The most common type of failures encountered, the initialization work, occurs when initialization code is changed. Some methods are only called at startup to initialize something in the application. The rest of the application will often depend on the initialization work having been performed. Even if the initialization method is changed, the change in effect will not occur because the method will never be called again. One example of where this occurred during the tests was in an update between version 2 and 3. At startup of versions 2 and below textures were created for the letters later used in the gameplay on letter blocks and guesses. In version 3 a new type of texture was introduced for the letters that are incorrect. Even though the update script modified the initialization method to include the generation of these textures, the changed method was never called. During a game, if guessing an incorrect letter the application will get a failure because the texture is not found.

To fix the failures created by initialization work not being performed again a call to the initializing method can be included in the update script. Unfortunately, this will not work in all cases because some initialization code is asynchronous. If the asynchronous parts of the initialization have not executed before code that assumes completion of the initialization it can lead to failures. In some cases this can happen because of a race condition in the original code that is not detected during regular execution. An example of this would be if the value set by the asynchronous code is used a relatively long time after the asynchronous operation begins. If the asynchronous operation starts at the start screen and the value is used on the game screen the probability that the asynchronous operation have finished before the user has managed to transition to the game screen is very high. However, if the asynchronous operation is started just before entering the game screen, as can happen when it is part of the update script, the probability of the instruction ordering being wrong is much higher. In these cases, the fault is not actually in the update script, but the update script can turn a potentially acceptable risk of failure into an unacceptable risk.

In other similar cases, the fault might actually be in the update script itself. Due to JavaScript being single-threaded, there can be some determinism in asynchronous behavior. It is possible that the original code making the call to the initialization code makes sure that the asynchronous code will be handled by the event loop before any code that depends on it. In this scenario the update script introduced the failure.

For the asynchronous failures that occurred, they were not further analyzed to determine if the update script introduced them or not. They are all presented as failures in this chapter.

### 5.2.3 Change in storage structure

The least frequent type of failure that occurred had to do with changes in storage structure. In order to save progress between restarts, the application saves some data to the main storage. This data is saved and loaded by the application when needed. Because the update script only
Dynamic updates of mobile apps using JavaScript

considers the changes in the code and not in the input or output a manual update script that transforms the saved data from the old version to the new version is needed. Both failures that were encountered during the tests had to do with how current score and high score was stored. In the first situation, the name of the file used for storage had changed and in the other situation the stored format was changed.
Chapter 6

Discussion

This chapter contains a discussion for both the results and the method used in the thesis. In 6.1 the results are discussed in terms of what we can conclude from them. In 6.2 the weaknesses and potential limitations of the method are discussed.

6.1 Results

6.1.1 Validation of update

When examining the results of the tests two questions related to the first research question have to be considered. The first is if the extra information provided by the change analysis helped in ensuring that the update was correct. The other question is if the tests, with or without extra information, give enough information to verify the correctness of the update. If a test fails it is obvious that something did not succeed in the update (assuming the test itself was correctly designed) but it is also important to consider if we can trust a passed test to tell us that the update was correct.

The results of the testing showed that the added information from the change analysis did not improve the amount of faults found. In fact, the information was so hard to interpret that it did not even provide enough details to allow for more tests.

This could to some degree have been avoided with better presentation of the results of the change analysis. Using the tool Beyond Compare presents information in a better way and with that tool a few more tests were identified. This applied despite the fact that Beyond Compare is not JavaScript-aware, simply treating the files as text files. A better presentation of the differences calculated with Simple_Tree_Matching can also be given, which was shown in the original paper describing the algorithm [24]. However, as far as I have been able to tell there is no such implementation available for JavaScript and implementing it myself was not possible within the scope of the thesis.

Even though using Beyond Compare provided some more tests it did not have any effect on the amount of bugs found. For this experiment it is possible to say that the extra information provided by Beyond Compare did not help in determining if the update was correct or not. The same conclusion about correctness was drawn without the extra information. One can also argue that the extra information helped, because more tests were performed and in another scenario these tests could have failed.

Assuming the failure types identified in the tests are representative for the failures that can occur information given to the developer should tell if initialization code, storage structure or non-configurable properties have changed. This is not achievable, at least not in an easy way, using the current change analysis. With that perspective, one can say that the extra information given by the change analysis was not useful for verifying correctness of the update. On the other hand, not all failures fit into one of these categories, so failures of different kinds can occur. I would consider the change analysis to have failed in regards to giving useful information to the tester because it did not give any information on where failures were likely to happen.
In regards as to whether we can trust a passed test to show that the update succeeded it can be a difficult question to answer. First and foremost, it must be considered that the usefulness of tests depend on the quality of the tests themselves. However, previous attempts at a formal and provable definition of correctness have been shown to not be enough to encapsulate the intuitive definition of correctness. For example, the reachability condition can be considered both too permissive and too restrictive. [7] Tests have been shown to be able to find faults in updates even when they are considered safe by some definition. [9] Judging by the previous research, the best option currently available is tests. Another argument for using tests to validate the update is that tests are often used to determine the correctness of an application. If the application that receives the updates is verified with tests it makes sense to verify the updates themselves using tests as well.

Using exploratory testing as the testing approach has both advantages and disadvantages. The main advantage is probably that exploratory tests have been shown to find a similar amount of defects as testing based on test cases, but in a shorter time. However, test case based tests have the benefit of repeatability. [11] When testing correctness of updates the same tests are performed for a multitude of update points. Due to tests being repeated, it can be seen as an argument for using a test case based approach instead of exploratory testing for the verification. This is an argument of convenience only, in regards to how valid the test result is the two approaches are similar. Exploratory testing reports fewer false defects [11] which might be an argument for exploratory testing actually being better for verification of correctness. Due to the advantage of repeatability with test case based tests, I would recommend using that approach to testing updates. However, no matter which approach is chosen the result is more or less equally valid.

Depending on how update points are chosen, there can be a very large amount of update points to be tested. In the tests that were performed as part of this thesis the same failure was often found using several different update points. This tells us that not all update points have to be tested to find all failures. In the work by Hayden et al. [9] they managed to eliminate from 87% to 95% of the update tests of the different programs they tested. With JavaScript being a dynamic language, it is harder to perform the analysis they performed for reduction of the amount of test points. In this thesis, an exploratory approach to selecting the update points to test was used. This seemed to work well in reducing the amount of tests needed. Given that most of the failures were found in more than one update point, it is possible that the amount of update points could have been reduced even further without decreasing the validity of the results. Even though I would recommend using test case based (and preferably automatic) testing for the updates it can still be a good idea to use an exploratory approach for selection of update points.

6.1.2 Types of failures

Three types of failures occurred that could not be handled by the automatically generated update script. Two of these (initialization, change in storage) could have been expected to require manual state transfer while one of them (non-configurable properties) is a result of the choice to make the update script JavaScript code that runs in the same context as the application. The drawbacks of this decision and alternative approaches are further discussed in 6.2, this part focuses on the other two types of failures.

The problem with the new version requiring a different kind of files was mentioned by Gupta et al. in their paper from 1996 [6] but they did not offer a solution to the problem. Bhattacharya and Neamtiu wrote in 2010 [3,4] about the need to support dynamic updates for several levels, including the database, in web and cloud application. In 2013 they wrote a paper together with three more authors [15] where they presented a way to provide on-the-fly schema updates for databases. However, no research on the more general problem with files have been found. It is probably impossible to provide a good automatic solution for this problem, because file formats
can be of any kind. Without additional information on the structure of the files before and after the update one cannot expect to be able to update the files automatically. In this case a solution without manual state transfer cannot be expected.

Problems caused by change in initialization work was not something that was found in the previous research. This is probably because most earlier research trust the developer to perform the state transfer and only limited consideration have been taken as to what this will mean for the developer. Kitsune [8] solves this problem by starting the application from the main method after an update, effectively restarting the entire application. This allows for the regular initialization code to run, before data migration and a jump to the appropriate point is performed (called control migration). Though this is a valid approach it requires more code to be written by the developer to decide when and how data migration and control migration is performed. For this thesis developer transparency was one of the goals, which meant a solution similar to Kitsune was not a good alternative. We must therefore try to determine if another solution could have been implemented that would not have these problems when initialization functions change or if manual state transition is required in this case.

The simplest transition to imagine if the initialization functions have changed is to call them again. This was shown to not work in all cases when the approach was attempted using manual update scripts. However, it works in some cases and would be a good idea to try out in an attempt at automatic state transfer. The problem then becomes to identify the initialization functions. A possible approach to solve that problem is to analyze the program code starting at the initial access point and determining which functions are called on only once in the beginning. This might be possible in some static languages, but is much harder in JavaScript due to its dynamic nature.

As mentioned earlier, even if initialization functions were correctly identified simply calling them in the update script would not result in complete correctness, manual state transition would still be required in some situations. The conclusion is therefore that manual state transition is probably required in the case that initialization functions have changed.

### 6.2 Method

It was decided that the updates were to be performed at the same level as the application itself. This is different from many other approaches at dynamic updates, where often the application is required to be running in a special environment or be compiled with a special compiler. There are several benefits with this, including that it is lightweight and takes advantage of the dynamic nature of JavaScript instead of building a different environment that is more dynamic. For languages such as Java and C which many of the previous attempts at DSU focus on, adding such a dynamic environment is necessary because the languages themselves are more static in nature. It is also good to have the update script expressed in the same language as the application. Because the developers already have experience in JavaScript, this makes the update script easier to read.

The main disadvantage of relying on the dynamic features of JavaScript is that some updates are made impossible. In ECMAScript 5 it became possible to modify property descriptors, i.e. deciding whether a property is enumerable, writable and/or configurable. Setting at least one of writable and configurable to false makes the property substantially less dynamic. This will cause the updates to fail as they depend on the default behavior where properties are both writable and configurable. Setting a property to non-writable will prevent the update script from changing its value. Setting a property to non-configurable results in it not being able to be deleted and it cannot be changed back to be configurable. In all cases that occurred in the tests this was possible to circumvent because the properties of the non-configurable and non-writable property object had the default behavior. However, it is not hard to find an example where a dynamic update in the same environment as the application would be impossible. One such example can be seen in Figure 6-1, where the value of a constant is changed.
This is a limitation of the method chosen that result in a worse performance in the flexibility goal of this thesis. However, I still believe that the decision to perform the updates at the level of the application was the correct one. First of all it has the benefits already listed above. Secondly, I would argue that as JavaScript is a dynamic language, it is bad practice to use features to make it less dynamic. Of course, there are situations where it can be appropriate, both the situation where the update failures occurred and the example of defining constants can be appropriate usages. In general, such situations should not occur too often. Even though such a failure occurred for four of the seven updates tested the cause of the problem was only two different non-configurable properties. These failures were also easily avoided with a simple fix that can be applied at all places where such failures occur.

The use of exploratory testing can be a threat to the validity of the results. Exploratory testing is known to be more dependent on the person performing the tests than test case based testing [12] [19]. When Itkonen et al. performed a replication [10,10] of their comparison between exploratory and test case based testing [11] they did receive similar results, despite the fact that the group of test subjects were slightly different. The groups were still relatively similar, but this shows that it is reasonable to assume approximately the same results for different testers, assuming they are reasonably similar.

For this thesis the tests were performed by me, i.e. a person with limited knowledge of the tested application but with extensive knowledge of how the update process works. With the goal of programmer transparency, it would have been more appropriate if the tests were performed by someone with knowledge about the application but limited knowledge of the update process. Unfortunately, this was not a possibility.

Tests were only performed for one application, which means it can be hard to generalize the results to other applications. It would of course have been preferable to test more applications of different kinds, but the time constraints for the thesis made this hard. With the application being relatively large and the updates being of varied types I would still argue that similar results can be expected for other applications.

The implementation was written to be as general as possible, but some details of the implementation is based on how the applications developed at Visiarc works. The ideas presented in this thesis should be applicable to any applications written in JavaScript or a similarly dynamic language but some of the details will have to be changed to fit in the domain the updates shall be applied. One example of such a detail is the inheritance described in 3.2.2.2.

I believe the research method chosen was successful because it provided good answers to the questions asked. While the same answers could perhaps have been reached by analyzing the
update process, by experimenting with a real application it is possible to say that the problems described will actually occur in an actual application.
Chapter 7

Conclusions and future work

The main conclusion from this thesis is that the dynamic nature of JavaScript is very valuable if one is to perform dynamic updates. In regards to the goals listed in 1.3 both continuity and programmer transparency can be considered to have been fully accomplished. With the decision to perform the updates at scene switches the perceived disruption in continuity is none. When switching screens, some transition time is already expected and the dynamic updates do not increase this time enough to be noticeable. Programmer transparency increases when the updates are written in the same language as the programmers already use. The only thing a programmer have to do with the current solution is to insert a call to the update service at an appropriate time. If a manual update script have to be written the script is written in a language already familiar to the programmer.

The two goals that this solution performed a little bit worse for was flexibility and correctness. Due to JavaScript having some features that makes it less dynamic there are some updates that are not possible with this approach. For correctness this thesis showed that many of the updates can be correct with an automatically generated update script, but just like previous research has shown this thesis also draws the conclusion that some state transition have to be written by the application developers.

7.1 Answers to research questions

In this section the answers to the research questions of the thesis are presented.

1. What kind of information about the change is needed to verify correctness of an update?

Based on previous research, the method that is most sensitive to failures in an update is testing. This thesis showed that there was a small set of categories of changes (described in the answer to the question about state transfer) that resulted in a failure in the update when using the method described in the thesis. It is therefore important to make sure that these categories of changes are thoroughly tested to verify correctness. An attempt was made to give helpful information to the tester about what changes occurred. Several different ways of presenting changes found by Simple_Tree_Matching were tested. The text based comparison tool Beyond Compare was also used as that tool presents changes in a more human readable fashion. None of these attempts resulted in any benefit for the tester as the same amount of failures were found without this information.

The information that might be the most useful is to get information about changes that can be categorized into one of the categories where failures were found. However, in the experiments these failures were found without knowledge of those changes, showing us that it is not strictly necessary to give the tester that information. Depending on the tester, the necessary tests are more or less likely to be performed anyway. With information about if any of the risky types of changes occurred the tester should be able to be more confident that the performed tests are enough.
Chapter 7 Conclusions and future work

2. How much of the state transfer can be automatic and how much is required to be specified by the programmer?

The results showed that only a few different types of changes required manual state transfer, at least for the type of application this thesis deals with. Three cases were identified where state transfer could not be automatic with the method described in this thesis. These cases were:

- If the data that is stored by the application have changed format. If this has happened between two versions a script that rewrites the old data to the new format have to be written as part of the manual state transfer.
- When initialization methods that are only called once when the application starts have changed a manual state transfer have to be written. This state transfer shall ensure that all initialization work have been performed before giving control back to the regular application.
- Depending on how the code is written, a change related to a property that is non-configurable or non-writable can require a manual state transfer. This problem is both specific to JavaScript and to how the update process was implemented. If a non-configurable property or non-writable property is modified directly it will be impossible with a manual state transfer. However, in some cases where the change is more indirectly related to the problematic property a manual state transfer that performs the change in an appropriate way can be written.

There was one failure found where the change that caused the problem could not be categorized into one of the categories. This shows that the three categories described above are not a complete description of where failures occur, but they cover most cases.

7.2 Future work

For future work my first recommendation would be to focus on how to detect changes that can be considered dangerous. This thesis found three change types that are considered dangerous in this context. Two of them are general enough to apply to most implementations of dynamic updates and the recommendation is therefore to focus on them. To detect changes in the data stored by the application two approaches can be considered. One is to execute the two versions of the application in the same way and compare what data is stored. This approach can be problematic because it is hard to decide what is meant by executing the two versions in the same way. An alternate approach would be to analyze all I/O calls to see if data is used in a different way between the two versions, which would indicate that the format of the data have also changed. To detect changes in initialization methods the focus should be on detecting which methods are actually initialization methods. With a successful detection of initialization methods it is relatively easy to detect if initialization methods have changed.

The update points that were tested were selected using an exploratory approach similar to how correctness was tested. This was natural to do because it fit in with the rest of the procedure for validation. Even though arguments are made in the thesis that this was a good decision, it would still be interesting for future research to compare the results when using an exploratory approach in the selection of update points and performing tests for all possible points of update.

The program Beyond Compare was used to display change information because the results from the Simple_Tree_Matching algorithm could not be presented in a good enough way. One interesting topic that could be examined for future work is whether a presentation similar to that of Beyond Compare, but based on AST comparison instead of text comparison, would be more useful for determining what should be tested.
References


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Simon Spetz-Nyström