Programming languages from a GIS point of view

Aarne Aramaa
Tartu University
Institute of Computer Science

Abstract

In developing geographical information systems (GIS for short) one must decide between various programming and scripting languages. Different languages come with different levels of support (or modules) for GIS related development. By comparing different languages and their capabilities from a GIS point of view we can find the strengths and weaknesses of each. Knowing what is missing from any language (or module) can help specify, for future development, what is needed.
1 Introduction

Geographic information systems are like any other information system, with a specificity that it does computations with spatial data. In software development, generally, different programming languages can be chosen by their different qualities (the platforms it can be run on, is it object oriented, standardized etc.). In developing geographic information systems, the decision between these languages should be made taking into account the existing possibilities of processing geographic data. Although small problems that are few in number can be solved without the use of any premade functionality, larger systems most likely rely on various operations such as verifying geometric shapes, detecting intersections between spatial objects, translating from one coordinate system to another and calculating surface areas.

In this article we list several known programming languages and modules for them that have spatial capabilities built in, we try to make arguments for each of the modules and tools, highlighting what functionalities either of them have and thus what could they be used for the best. We also display empirical results of testing the different modules on similar data and equivalent functionality.

We also describe using the technologies in terms of creating an application, like a desktop application for performing analyses based on or related to spatial data, but
also, maybe most importantly, creating a web map application, and in doing so, we describe the interoperability between these tools.

2 Languages and their uses

Next we list a few programming and scripting languages that have found use in geospatial-related development.

JavaScript – the front-end part of a GIS, which mainly means visualization of data on a map, can be implemented in the popular scripting language JavaScript with the use of OpenLayers[1], a library specifically for mapping data comes with functionalities that allow the system to display raster and vector data in an efficient manner. Examples of other tools to use with JavaScript are Leaflet[2], Esri Api for JS[3], CartoDB[4], MapBox[5], Node.js[6] and JSTS[7].

Java has been used to create popular tools like GeoTools[8] and GeoServer[9] that come with APIs to help GIS development. GeoTools is an open source Java library with an intent purpose of helping manipulate geospatial data. GeoServer on other hand is a software server meant for editing and viewing geospatial data. For spatial operations Java can be used with JTS[10], a topology suite for just that. JTS has been ported to other programming languages as JavaScript (the already mentioned JSTS) and Ruby’s RGeo[11].

C++ being the most used language for desktop development, has been used to create tools such as QGIS[12] and MapServer[13], which similarely to the aforementioned GeoServer can be used for editing and publishing spatial data for geographic information systems. To enable manipulation of spatial data, C++ can be complemented with GEOS[14], which is open source.

Python is an overall popular language and so it is also popular within the GIS community. Python can be used within different GIS-related software as a scripting language. The aforementioned QGIS and GeoServer both allow the use of Python, along with other software like FME[15] and ESRI’s ArcGis suite[16]. To enable spatial calculations for Python, popular scientific packages like SciPy[17] and NumPy[18] can be used as they come with geometric objects and linear algebra related functionalities.

SQL – as databases are an important aspect of information systems, SQL can find massive use in having geospatial capabilities. An example of an addition to PostgreSQL is PostGIS[19]. And Oracle Spatial[20] for Oracle databases. Geospatial additions to SQL allow the user to execute spatial joins and selects of tables wherever geometric objects are within a specified relation.
3 Open Geospatial Consortium

The geospatial libraries of various languages conform to many standards set by the Open Geospatial Consortium (OGC) [21] to allow for separately developed GIS tools to be more interoperable. For this article we determine the interoperability of tools by the implementations of the standards. Some of the standards we observe that are set by the OGC are as follows:

GML – A geographic markup language, basically XML for geographic information. A GML allows the user to save for example geometric objects with its shape, coordinate system, relation to other objects (such as overlapping, containing or touching), directions, time-related data, etc.

WFS – Web feature service. The WFS serves as an interface that allows the user to query, spatial objects or features, for example counties drawn on top of a map image.

WFS-T – Transactional Web Feature Service. The WFS-T is an improvement of the WFS in terms that it allows the user to create, delete and update features in addition to just querying them.

WMS – Web map service. Similar to the WFS, the WMS serves as an interface for serving up map images with embedded data, referring to its geographic location.

WPS – Web processing service. The WPS provides an interface for using geoprocessing services, for example for inputting two geometric features and returning them joined as one feature.

WCS – Web coverage service. The WCS is an interface standard for retrieving data that might have different values at different locations or points in time, for example amount of rainfall.

SLD – Styled Layer Descriptor. SLD is a standard based on XML for determining the appearance of map and feature layers, like setting the colour of a type of area.

WMC – Web Map Context. WMC is a standard for saving a users current view, instead of loading multiple layers of data with the WFS every time the application is restarted. Also allows for sharing views between users.

FES – Filter Encoding Specification. The FES is another XML based specification for selecting and filtering features based on some attribute.

These standards allow for greater interoperability, for example a Java application could save a geometric object into a GML, send a request to a C/C++ service to modify it (for example create a 50m buffer around it), and then receive a resulting geometric object as a GML which could be then parsed and used further in the Java application.
4 Comparison of tools

In this chapter we compare the tools grouped by their roles in the information systems.

4.1 Server tools

<table>
<thead>
<tr>
<th>Name</th>
<th>WMS</th>
<th>WFS</th>
<th>WPS</th>
<th>WFS-T</th>
<th>WCS</th>
<th>WMC</th>
<th>SLD</th>
<th>FES</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoServer</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>MapServer</td>
<td>✔</td>
<td>✔</td>
<td>O</td>
<td>O</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>QGIS Server</td>
<td>✔</td>
<td>✔</td>
<td>O</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td>ArcGIS Server</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>O</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Symbol       Meaning
✔ Supported by standard version
O Supported through an (external) add-on
X Not supported

Above are the benchmarking results of Open Source Geospatial Foundation’s (OSGEO)[22] 2011 benchmarking competition describing how the open source servers: MapServer, GeoServer and QGIS compare performance wise when dealing with vector data. The most surprising may be the C-based QGIS Server and its comparably low performance.
Above are the results for vector data, without the results for GeoServer, as they opted to participate only in the vector data benchmarking. Visibly there is almost no difference in how the tools compare to each other.

According to Geotests.net’s 2014 benchmarking results, the differences between MapServer, GeoServer and QGIS server are:

- QGIS does not allow for coordinate system reprojections, however in GeoServer and MapServer this can be done with the help of add-ons.
- QGIS and MapServer do not allow the use of a transactional web feature service.

All three allow for the use of Web Feature Services, Web Map Services, Web Coverage Services, parsing GMLs and GeoJSONs and cartographic analysis.

### 4.2 Web client tools

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>WMS</th>
<th>WFS</th>
<th>GML</th>
<th>GeoJSON</th>
<th>GeoRSS</th>
<th>KML</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenLayers</td>
<td>JS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Leaflet</td>
<td>JS</td>
<td>✔</td>
<td>O</td>
<td>X</td>
<td>✔</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>ArcGIS API</td>
<td>JS</td>
<td>✔</td>
<td>✔</td>
<td>O</td>
<td>O</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>CartoDB</td>
<td>Ruby/JS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>O</td>
<td>✔</td>
</tr>
<tr>
<td>MapBox</td>
<td>JS</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>Supported by standard version</td>
</tr>
<tr>
<td>O</td>
<td>Supported through an (external) add-on</td>
</tr>
<tr>
<td>X</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
4.2.1 Front-end code examples

We compare the open-source libraries OpenLayers and Leaflet and their code to visually present one of the main differences between them – simplicity.

```javascript
// Background raster map layer
var raster = new ol.layer.Tile(
  source: new ol.source.MapQuest({layer: 'sat'}))

var source = new ol.source.Vector();

// Vector layer to draw on
var vector = new ol.layer.Vector(
  // Set feature fill and stroke color etc
);

// The map object to display on page with initial pan and zoom
var map = new ol.Map(
  layers: [raster, vector],
  target: 'map',
  view: new ol.View(
    center: [-11000000, 46000000],
    zoom: 4
  )
);

var typeSelect = document.getElementById('type');

// Variable for new feature and function to handle drawing
var draw;

function addInteraction() {
  var value = typeSelect.value;
  if (value !== 'None') {
    draw = new ol.interaction.Draw(
      source: source,
      type: '/* @type {ol.geom.GeometryType} */ (value')
    );
    map.addInteraction(draw);
  }
}

addInteraction();
```

This is an abridged example of the javascript file, using OpenLayers, for an application that displays a background map and allows the user to draw polygons on top, without any updating of databases or processing etc.
This is javascript code for an analogous application using Leaflet.

In both of these examples a map viewer, map layer and an overlay layer for drawing the features are roughly defined, for them to be working implementations, a toolbar or active tool should be defined and they should retrieve raster maps from a back-end mapserver.

The main difference between Leaflet and OpenLayers is the weight. Leaflet is a simple and light-weight library, while OpenLayers is obviously more complicated it allows for more functionalities. For simple map applications, anything like displaying a few locations or small amounts of location related data on a map for a website could be recommended to be done using Leaflet, reserving OpenLayers for when more extensive applications are created. OpenLayers was initially released on 2006, while Leaflet was released in 2011, hinting at the fact that leaflet might catch up to OpenLayers in the functionality department.

According to geotests.net’s 2014 benchmarkings, OpenLayers and Leaflet differ in Leaflet not utilizing cache. Other tools listed, that are on par with OpenLayers in terms of the tested functionalities are GeoEXT, which is OpenLayers with Ext.js (a JavaScript framework interoperable with jQuery and utilizing AJAX for creating interactive webapps) and MapQuery which has not been updated since 2012.

The popularity of the APIs seems varied with around 4000 questions tagged with OpenLayers and around 1000 questions for Leaflet on gis.stackexchange.com, around 6500 questions for both on stackoverflow.com, around 3000 repositories related to Leaflet and 300 related to OpenLayers on GitHub.
5 Interoperability of tools

In the following chapter we explain by example how some of the mentioned tools can be used together in creating geographic information systems.

5.1 Interoperability in web applications

When creating a web application, the tools could be split into three categories: back-, middle- and front-end.

The above figure is to illustrate possible combinations between the tools of different layers.

In most cases the features to be displayed on the map would be held in a database, be it PostGIS or Oracle Spatial. The features would be geometric objects with a list of coordinates and the respective coordinate system. These features could be queried as a Well-Known Text (WKT) format and parsed into a GeoJSON in the application’s back-end or the application server software to be forwarded to the user interface through a HTTP GET.
6 Conclusion

There are multiple widely used GIS tools on any level for open source development and otherwise, which allows developers multiple choices. It may seem that a lot of the decision boils down to personal preference, by which we mean taking into account the platforms in use (for example an already implemented Java web application would most likely mean that the GIS part of it would be implemented in Java as well). There are not many GIS-specific limitations as to what languages can be used where.

For successfully implementing a GIS application, knowledge of Python, JavaScript, SQL and either C/C++ or Java come in handy or even essential depending on the level of the application.
References
1. OpenLayers
   openlayers.org
2. Leaflet – A JavaScript library for interactive maps
   leafletjs.com
3. Esri API for JS – ArcGIS API for JavaScript
   developers.arcgis.com/javascript/
4. CartoDB
   cartodb.com
5. MapBox
   mapbox.com
6. Node.js
   nodejs.org
7. JTS – JavaScript Topology Suite
   github.com/bjornharrtell/jsts
8. GeoTools
   geotools.org
9. GeoServer
   geoserver.org
10. JTS
    vivid solutions.com/jts/JTSHome
11. RGE0
    github.com/rgeo/rgeo
12. QGIS
    qgis.org
13. MapServer
    mapserver.org
14. GEOS
    geos.osgeo.org
15. FME
    safe.com
16. ArcGIS Suite
    esri.com/software/arcgis
17. SciPy
    scipy.org
18. NumPy
    numpy.org
19. PostGIS
    postgis.net
20. Oracle Spatial
    oracle.com/us/products/database/options/spatial/overview/
21. Open Geospatial Consortium
    opengeospatial.org
22. OpenSource Geospatial Consortium
    osgeo.org
Survey of Code Clone Detection Techniques

Jaanus Jaggo,
University of Tartu, Estonia

Abstract. Code clones in software system are often considered as an error-prone practice. During recent years many clone detection techniques have developed to help developers organizing their system. This paper we describe the overall code clone taxonomy and major approaches in code clone detection. We have also selected 5 different implementation of major techniques that we will observe in detail. We will also compare those techniques based on their functionality and efficiency. Overall there is no one best solution for clone detection but rather a lot of alternatives developed for different problem scenarios.

Keywords: software clone, clone detection techniques, comparison

1 Introduction

Copying and adapting code sequences is a common process in the software development. This practice results in similar sections in the source code, called code clones. Code clones are often intentional and even part of the program architecture, but other times there can be accidental clones that developer is not aware of. In a good software system all intentional clones should be documented but studies show that they rarely are [1]. If a clone gets forgotten there is always a danger that during the software development cycle a change or a fix got applied on only one appearance of the clone leaving the other place faulty. Finding those kind of clones and merging them or guiding the developer during maintenance is the major reason for code clone detection.

Code clone detection is also useful for detecting plagiarism or intellectual property theft and even for extracting code patterns from the source code. However depending on the exact problem domain there are different clone detection techniques. For example the one used in plagiarism detection might not be practical in large scale system maintenance and vice versa.

In this article we are giving an overview of different tools and techniques used in code clone detection investigating how they are made of. We are also comparing them from the perspective of functionality, performance and simplicity. At the end of the paper there is a summary table about observed tools.

1.1 Classification of code clones and detection techniques

A widely used notation of [2] divides code clones into three classes:

1. Type 1 clone is a textural copy where only formatting, comments and whitespaces may differ.
2. Type 2 clones are such where also parameters or variable names can be different.
3. Type 3 clones are such where one or more statements can be inserted or deleted and statement locations can be changed.

In software development industry type 1 clones are rare and detecting those will be just partially useful. Such tool should detect at least type 2 clones and even type 3 clones if they are still similar enough. On the other hand for plagiarism Type 1 and 2 clones are often sufficient enough and in some cases two submissions could be classified as Type 3 clone even if they are not copied.

The variety of different clone detection techniques is obviously larger than different clone types but in general they can be divide into three classes corresponding to those types.
1. Text or Token based clone detection tools can mainly discover type 1 clones but with some syntactic enhancement they can also detect type 2 clones.
2. Syntax tree based clone detection tools can detect type 1 and type 2 clones.
3. Semantic based clone detection tools can detect all three types of clones.

1.2 Comparison of clone detection tools

Beside clone detection technique and clone types the tool can detect there are a lot of other characteristics used for comparison of clone detection tools. Some of them listed in [3] are interesting in our work as well:

- **Language support** – set of particular languages supported by the tool
- **Clone granularity** – indicates the granularity of reported clones. The clones could be reported without any syntactic boundaries or they can be fixed within a method, block etc.
- **Worst case computational complexity** – it gives an overall estimation of how well the tool scales up.
- **Pre-/Post-Processing** – if any special pre or post-processing technique is applied during the clone detection.

2 Wavelets based clone detection

A naïve method for finding clones from two text files would compare each character with every other. Obviously this approach would not scalable at all. Wavelets are certain mathematical functions allowing to transform data into a smaller form. Kilgi [4] has used wavelets to reduce the number of characters in the source code and apply per character clone detection in large projects. For example Linux kernel contains 3 666 796 SLOC (source lines of code) but it can be reduced to just 57 293 characters by applying wavelet transformation 6 times, each time reducing the size of the data by 2.

Kilgi’s tool has four major steps, each implemented as a different program. These steps are:

1. **data transformation to numeric form** – this step gathers all the files in the project directory and removes all whitespace characters, however comments are
maintained. This initial processing reduces the data about 33%. After that every string is converted into a sequence of ASCII codes.

2. **discrete wavelet transformation** – this step applies Haar wavelet transformation [5] to the data. Haar transformation finds a pairwise average between each two consecutive value in the sequence starting from the last element. Each result is rounded to two decimal points and put to the output sequence. This process is repeated multiple times reducing the number of characters by half in each level. The whole process is also repeated with different offsets which grants better tolerance for false negatives. Both the maximum level and the number of offsets can be specified by a program parameters.

3. **match detection** – characters are matched based on the longest common substring algorithm starting on the highest level with the least number of characters in it. Whenever a match is detected the algorithm proceeds to lower level to confirm it is not a false positive. Finally only matching sequences of a certain size are accepted as clones. However it can happen that a clone pair has just a small modifications made on one individual, for example identifier names changed. For this occasion the program allows a small gap between two sequences and merges them together to a single clone.

4. **clone representation** – clones detected are reported in a HTML file for human validation.

2.1 **Case studies**

The performance test was carried on Linux kernel, which has 141 361 SLOC containing 3 666 796 characters. The whole process took just about 15 minutes to complete on a single processor core when 7 offsets was used. Because of the simplistic nature of the algorithm it is easily scalable to multiple cores as well and would theoretically perform a significantly better on a GPU. Compared to other purely text-based clone detection solutions, namely NiCAD 3.5 and Simian 2.3.35 the current tool performed slower but detected more clones. The comparison of those tools is shown in the table 1.

<table>
<thead>
<tr>
<th>Detector</th>
<th>Total execution time</th>
<th>Clones found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wavelets</td>
<td>896.7s</td>
<td>1449</td>
</tr>
<tr>
<td>NiCAD</td>
<td>19.3s</td>
<td>152</td>
</tr>
<tr>
<td>Simian</td>
<td>6.5s</td>
<td>262</td>
</tr>
</tbody>
</table>

The overall the precision (amount of false positives) and recall (amount of undiscovered duplicates) of a purely text-based clone detector is definitely lower compared to following tools described below. However the current technique is completely language independent which cannot be achieved with any syntax driven approach. This is not just convenient but also practical as the current tool can find clones even if they are hardcoded in another language, for example from SQL statements in Java code.
3 Token based clone detection - CCFinder

CCFinder (Code clone finder) [1] is a token based clone detection tool, that was designed for detecting clones from huge legacy software system, which has one million lines of code and is written in both COBOL and PL/I-like language. The tool has three main phases. At first lexical analyzer produces a token sequence from the input code. Then during the transformation phase a language-dependent component translates the code into an uniform format. This component is quite lightweight and is therefore replaceable for different languages. It has a set of language specific transformation rules for example in C++ code all single statements after if-statement are moved under a compound block so following if-statement if (a == 1) b = 2; is transformed into: if (a == 1) {b = 2;}. Also during this phase all identifiers, variables and constants are replaced with a special token which allows code segments with different variable names to become clone pairs. Right now CCFinder transformation component supports C, C++, Java and COBOL languages. Finally from all the substrings equivalent token sequences are detected and marked as clone pairs. It uses a suffix-tree algorithm with both space and time complexities $O(m \times n)$ where $m$ is the maximum length of involved clones and $n$ is the total length of the source file. For optimizations they have implemented following techniques:

1. **Alignment of Token Sequence.** Because the clones that begin at the middle of a statement are not particularly useful. Only specific tokens at the beginning of clones are allowed. Some of those tokens are for example block opening keyword “;”, selection statements (if, else, switch) and declarations (class, enum, typedef). While this restriction might slightly reduce the sensitivity of the algorithm, it helps to reduce the number of nodes in the suffix tree to one third of its original size.

2. **Repeated Code Removal.** Sometimes, especially when dealing with case-statements, there might appear intentional repetition of a short code segments. When the second continuous sequence is found it can be extracted and all following sequences from the suffix tree can be thrown away.

3. **Division of Large Archive of Source Files.** If the total size of source files exceeds the memory limit for the suffix tree, the tool automatically employs a “divide and conquer” approach. While this technique has a quadratic complexity in worst case with right implementation it was still really efficient even for large code files up to 10 million LOC.

3.1 Case studies

The case study was conducted on Pentium III 650MHz and 640MB RAM PC. For JDK 1.3.0 with about 570k LOC in 1877 files. The execution time was about three minutes. They also tested the usefulness of their transformation rules and it turns out that if the transformation rules, parameter replacement and repeated code removal were disabled then only half of the original clone pairs were detected. Another case study was conducted on systems developed by two companies, one filed a lawsuit against another suspecting an illegal code theft. Both systems consists of more than 100 C files and the their coverage value was over 50%. The result was filed in a court as an evidence.
4  Abstract syntax tree based clone detection

While token based clone detection is fast, it has a little knowledge about the program syntactic structure and therefore may produce too many false positives. This leads to the more syntactic clone detection methods. By comparing ASTs we can avoid purely lexical clones like the ones that might start and end in the middle of a statement. Comparing ASTs for clone detection is in principal the same as comparing each sub-tree to each other sub-tree in the AST.

The major problem of this approach is that it has $O(N^3)$ complexity where $N$ is the number of AST nodes. The naïve solution obviously does not scale well. Baxter [6] has tackled this problem by partitioning the sub-trees into $B$ buckets allowing to compare only sub-trees within the same bucket. In practice if $B$ is chosen in the same order of $N$ the comparison complexity turns out to be close to a constant. Therefore the overall computation can be done approximately within $O(N)$ time. The question is how to choose such distribution function that would not make too many false negatives. Regular hash function would be useful for detecting exact clones only. Baxter solved this problem by using an artificially bad hash function. They had observe that near-miss clones often differentiate by just small changes in the leaves of the AST. Based on that observation they made a hash function that ignores small sub-trees and drops such trees into the same bucket.

In the comparison phase we would like to compare sub-trees based on their similarity instead of exact equality. This comparison is obviously done within the same bucket. The similarity of two sub-trees is defined as:

$$\text{Similarity} = \frac{2 \times S}{2 \times S + L + R}$$

Where:
- $S =$ number of shared nodes
- $L =$ number of different nodes in sub-tree 1
- $R =$ number of different nodes in sub-tree 2

4.1  Clone sequences

Previous described method is good for detecting clones purely on trees, however there are a lot of sequences involved in programming language semantics such as sequences of statements or function argument. These sequences are not just arbitrary ASTs but rather left- or right-leaning trees which are produced by applying left or right recursive production rules during parsing phase. Each assignment operation on figure 1 is dependent on previous operations and special treatment is necessary to find clone sequences from such tree.
Baxter’s sequence detection algorithm compares each pair of sub-trees containing sequence nodes, looking for maximal length sequences in previously detected clones. Detected sequences are stored in a list which allows to get hash code for subsequences very quickly. Overall this method can detect sequential clones where sequent elements are in different order but miss clones where statements have been inserted or deleted.

4.2 Generalizing clones

After initial exact and near-miss clones are detected another loop visits the parents of already detected clones and checks if the parent is a near miss clone too and discards its sub-clones. After all each near-miss clone is just a composition of exact clones therefore expanding an exact clone this way will increase the clone coverage even further.

4.3 Case studies

The system was applied to a 7 years old process control system having approximately 400 KSLOC of C code. There were 15 programmers working on it at this time and none of them were one those who developed the original system. The average clone percentage identified by current system was 12.7%. Most of the clones were relatively small ones about 10 SLOC while clones larger than 25 SLOC were quite rare. However an unexpected clone of size 497 SLOC was also discovered from this system. While the system could process 100 KSLOC code within 120 minutes the main bottleneck was memory instead because the algorithm relies on RAM-based hash buckets. To overcome this problem they had to divide the original 400 KSLOC project into 100 KSLOC junks which were fitting it into the 600Mb RAM available.
5 Abstract syntax suffix tree based clone detection - CPDetector

Baxter et al. used hash function and inexact matching based on similarity metric to speed up the comparison. However, this approach affected the precision and even if the partitioning approach helped a lot, the comparison of sub-trees within the same bucket was still pairwise and hence it has a quadratic time complexity. On the other hand, token-based analysis can run almost linear speed like CCFinder did but it has its own problems, for example, it tends to report clones between two different functions which would not be useful at all.

Rainer’s approach [7] enhances AST based clone detection by mixing it with a suffix tree detection algorithm that is commonly used in token-based detection tools. Their method consists of 4 steps:

1. Parse the program and generate AST.
2. Serialize AST.
3. Apply suffix tree detection.
4. Decompose resulting token sequences into complete syntax units.

The first step is just a common procedure. In the second step, a preorder traversal algorithm is applied that translates the syntax tree into an isomorphic token stream that maintains the syntactic structure of the original tree. For an example, for a program in Listing 1, a corresponding preorder serialized sequence in Listing 2 is produced.

```plaintext
If c + y then a := j; else foc; end if;
If p then a := j; else foc; end if;
If q then x := k; else bar; end if;
```

Listing 1. Example sequence of if statements in Ada.

```
Seq23
   If $id_0 + id_0 id_0 = id_0 id_0 call_1 id_0$
   If $id_0 = id_0 id_0 call_1 id_0$
   If $id_0 = id_0 id_0 call_1 id_0$
```

Listing 2. Serialized listing for the program in Listing 1. The postfix number of tokens shows how many children the corresponding sub-tree has.

The clone detection itself is based on a string-based suffix tree algorithm by Ukkonen [8]. At this stage, the actual name of identifiers and literals is ignored; they are used later for distinguishing between type-1 and type-2 clones. Finally, the token sequence clones are decomposed back to syntactic units. For our example in Listing 2, the longest clone sequence is shown in Listing 3 but it’s not a syntactic sequence any more.

```plaintext
id_0 = id_0 id_0 call_1 id_0
```
Listing 3. Longest clone sequence in for Listing 2.

During decomposition the clone in Listing 3 is separated into three single syntactic units: \(<\text{id}_0\>\), \(<\text{id}_0 \ \text{id}_0\>\) and \(<\text{call}_1 \ \text{id}_0\>\). These units can be used to match the clone sequence against the original AST and even used to filter clones based on various criteria such as length, type of clone and syntactic type.

5.1 Case studies

For evaluation the author used technique similar to Bellon benchmark which is a common quantitative comparison method for software clone detectors [9]. The current technique is reported to found 71% more clones compared to CCFinder and almost double those found by other AST-based tools. For performance comparison they had a 64bit Intel dual-core (3.0 GHz) setup with 16 GB RAM running on Linux where only one CPU was used. Detecting clones from PostgreSQL database (253k LOC) took 2 hours and 45 minutes. Overall their system offers a scalability comparable to token-based techniques but is better at finding type-2 clones.

6 Semantic based clone detection

Modification that disturb the structure of the code will make the clone more complicated to identify. Some of those modifications are:

- Parts of code that are executed under different circumstances
- Expressions and/or variables are changed
- Parts of the code are inserted or deleted
- Parts of the code is moved to the different location

To identify such modifications text or syntax based clone detection tools have to do tradeoff between precision and recall. To detect not identical clones these tools have to ignore certain properties which may lead to false positive results. Krinke [10] has developed a semantics based approach which does not suffer under this problem. It represents a program as a fine-grained program dependency graph within it identifies similar sub-graphs and maps later maps those back to the user for presentation.

6.1 Fine-grained Program Dependence Graph

The traditional program dependence graph (PDG) [11] is a directed attributed graph. Vertices of this graph are representing assignment statements and control predicates. The edges are representing the dependences between program components and they are divided into control and data edges. In Krinke’s PDG the AST vertices have almost one to one mapping onto PDG vertices. It has also specialized edges called immediate (control) dependences which are evaluated before rest of the source and for dataflow information it has value dependence edges and reference dependence edges, the difference between these two is that in case of reference dependence edges the value is
stored into a variable. Figure 2 shows an example of fine-grained PDG for the code in listing 4.

```c
void f (int a, int b, int c) {
    x = a * (y = b + c);
    z = x + y;
}
```

**Listing 4.** Example code for PDG [7].

![Fig 2.](image)

Fig 2. Fine grained PDG for code shown in Listing 4 [7].

In this example we have an entry vertex 5. Vertices 6, 7, 8 are formal-in vertices and 13, 14 and 15 are formal-out vertices. The vertex 12 is a compound vertex that groups the sub-graphs together. All other vertices are expression vertices between different operators. The benefit of this representation is the ease of the identification of similar or identical vertices and edges.

### 6.2 Finding similar subgraphs

Two graphs are isomorphic if every edge is bijectively matched to an edge in the other graph and the attributes of the edges and the incident vertices are the same. In general checking graph isomorphism is NP-complete problem. Moreover in practice we are interested in finding similar sub-graphs not just isomorphic sub-graphs. By Krinke’s
definition graphs $G$ and $G'$ are similar, if for every path $v_0,e_1,v_1,e_2,...,e_n,v_n$ in one graph there exists a path $v_0',e_1',v_1',e_2',...,e_n',v_n'$ in the other such that the attributes of the vertices and the edges are identical if the paths are mapped against each other. Also there is a restriction that all paths have to start at a single vertex $v \in G$ and $v' \in G'$ such that $v_0 = v$ and $v_0' = v'$ for all such paths.

Kirke’s approach constructs a maximal similar sub-graph by induction from starting vertices and matches length limited similar paths. Instead of checking every possible pair, an inductive approach allows to match an edge under consideration to the multiple places at once. However even with this “smarter” approach the complexity is still $O(|V|^2)$. Therefore to make it feasible only a subset of starting vertices should be considered assuming that other vertices are reached during the construction process. The procedure that finds suitable subset of starting vertices should be based on specific features of those vertices which turns out to be highly application specific. One such possibility is to use entry vertices only allowing to find similar procedures. However Kirke decided to use predicate vertices instead because then it’s possible to find similar pieces of code independent of procedures. Therefore for every pair of predicate vertices the maximal similar sub-graphs are generated.

The problem with this kind of selection is that sometimes identified similar sub-graphs could represent structural properties only that we already had in semantic analysis based approaches. It is because most of the data dependencies could not match and then mainly control dependence edges are included in the sub-graphs. The solution is to assign a weight for each sub-graph corresponding to its data dependencies. A simple criterion is just to use the number of data dependence edges in this particular sub-graph as a weight.

6.3 Case studies

In the current article they have not mentioned the specification of the hardware system they were using. Their tool was tested on various C projects with about 2000 to 25000 LOC. Due to the pairwise comparison the overall system has a quadratic complexity but the exact running time is dependent on the size and the amount of similar sub-graphs in the program. For this reason a k-limited approach was adapted where the k is the number of nodes in the maximal path constructed while constructing the sub-graphs. The tradeoff here is the ability to detect larger clones. It turns out that while smaller program could be analyzed with k-limit up to 100 then one of their case having too many clones could not run for k-limit bigger than 25 because the running time was already 46 hours. From the positive side among all duplicates checked there were no false positives (100% precision). They also claimed that a k-limit around 20 is ideal for sufficient recall for example it took just 47 seconds to analyze bison under this limit.

7 Conclusion

In this paper we observed five different code clone detection techniques and composed a summary table in appendix 1. While there are more different clone detection
techniques invented, they are mainly modifications or extensions of these five. Four out of five represented tools can be easily classified as a text/token based, syntax based or semantic based technique. Syntax suffix tree based approach [3] on the other hand was a composition of both token based and syntax based approach. In the future there will be even more hybrid clone detection techniques combining the advantages of two or more techniques. For example it was stated [2] that the AST based detection can be converted for dataflow graphs as well without any modification of trees.

References

## Appendix 1: Comparison table

<table>
<thead>
<tr>
<th>Technique</th>
<th>Language support</th>
<th>Clone classes</th>
<th>Granularity</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Problematic structures</th>
<th>Computation complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wavelet based</td>
<td>All languages</td>
<td>type 1 &amp; 2</td>
<td>character</td>
<td>efficiency, language independence</td>
<td>lower precision and recall</td>
<td>long non-similar gaps</td>
<td>$O(n \times \log(n))$</td>
</tr>
<tr>
<td>Token based</td>
<td>C, C++, Java, COBOL</td>
<td>type 1 &amp; 2</td>
<td>syntactic unit</td>
<td>efficiency</td>
<td>needs language specific tweaking</td>
<td>nodes inserted, deleted and modified</td>
<td>$O(m \times n)$</td>
</tr>
<tr>
<td>AST based</td>
<td>C, C++</td>
<td>type 1 &amp; 2</td>
<td>syntactic unit</td>
<td>efficiency</td>
<td>recall, (lot of assumptions)</td>
<td>nodes inserted and deleted</td>
<td>$O(n^3)$ in worst case, $O(n)$ in practice</td>
</tr>
<tr>
<td>AST suffix based</td>
<td>C</td>
<td>type 1 &amp; 2</td>
<td>syntactic unit</td>
<td>efficiency</td>
<td></td>
<td>nodes inserted, deleted and modified</td>
<td>$O(m \times n)$</td>
</tr>
<tr>
<td>PDG based</td>
<td>C</td>
<td>type 1, 2 &amp; 3</td>
<td>function</td>
<td>100% recall</td>
<td>implementation complexity, Efficiency</td>
<td>none</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

* In the formula n denotes the number of nodes in the program and m denotes the maximum number of nodes in the largest clone sequence.
Similar code detection

Viktor Karabut

December 10, 2015
Contents

Introduction 3

1 Similar code detection 4
   1.1 Description of ACT .................................................. 4
   1.2 Translation ACT ....................................................... 5
   1.3 Definition of similarity .............................................. 7

2 ACT search engine 11

3 Conclusion and Future Work 12

References 13
Abstract

Goal of this work is to develop a software system that finds similar code for specific Java code fragment. During this work we use convert code into intermediate representation - ACT (approximate code tree) and give a formal definition of code similarity in terms of ACTs. Using this definition we can compute how similar two Java code fragments in scale from 0 to 1. However one to one comparisons requires quadratic time and doesn’t suitable for big code bases. To improve search performance we implement ACT search engine. Search engine hashes ACT and searches for similar code in the database of fingerprinted, versioned and indexed ACTs.

Introduction

Supreme goal of my work is an implementation of IDE plugin which check written code and gives suggestion about code organization. For example some similar routines from different modules of large project can be refactored into one utility method. Another usage scenario is a code fragments which can be replaced with utilities available from some existing library. Often code reuse is limited by legal issues, such as patents, copyrights [1]. However there is a wide variety of an open source projects, from where code can be used as an inspiration or even be directly used.

Our goal is to find not intentionally copied code, but code fragment which is similar because they implements similar algorithms or similar code routines. In many previous works [2][3] code clones considered as textual copies of code with changed formatting, added comments or renamed variables. In this article we emphasise on semantically similar code, which cannot even have similar statements, but which does semantically same work. For example for every while-cycle we can write semantically equivalent while-cycle or replace if-else with switch-case statement.

In first chapter we consider questions about code similarity. In order to detect similar code fragments we will parse Java language code into AST. After that we introduce intermediate representation of parsed code: Abstract Code Trees (ACT). During translation to ACT we get rid of redundant for our purpose information.

In second chapter we will consider fast search engine for code fragments. ACT each-to-each comparison complexity is $O(n^2 \cdot m^2)$ where $n$ is number of trees and $m$ is size of each tree. For large code base performance of quadratic algorithms is not feasible. To solve this issue we use a fingerprinting approach. Fingerprint is a generalized version of ACT, where removed some information. Fingerprinting allows to find in constant time get some relatively small set of matching ACT from where we can find needed code fragments using a slow quadratic algorithms.
1 Similar code detection

Simplest possible way to find code duplication would be line-by-line comparison of source lines. However code can differ by an indentation, comments, variable names.

Other approach is parse language construct into language-specific Abstract Syntax Trees (AST) and then compare each node by node using recursive similarity definition, as it did Baxter in his work [2]:

\[ \text{Similarity} = \frac{2S}{2S + L + R} \]

where S is a number of shared nodes, L is a number of different nodes in left subtree, and R - in right subtree.

In this work we will use similar approach, but instead of working with language-specific AST we will translate it to language-agnostic intermediate code representation: Abstract Code Trees (ACT). ACT is a digest of program syntax structure. Main reason of introducing ACT is to give formal definition of 'code similarity' which allows us to reason about effectiveness of our search methods.

Intermediate representation will allow to make language-agnostic search engine, so it would be possible to migrate to another language. Laconic by nature ACT will help to keep the search engine simple and effective.

As a bonus at Java translation step we get rid of various Java 'sugar' constructs, such as introduced in Java 6 'foreach' cycle for collection traversing [4].

1.1 Description of ACT

ACT (Abstract Code Tree) is an intermediate representation of code. ACT structure was inspired by Nielson’s while language [5]. ACT tree structure are describes these Extended Backus-Naur Form (EBNF)[6] rules.

\[
\begin{align*}
\text{stmt} & = \text{block} | \text{if} | \text{while} | \text{expr} | \text{try-catch} | \text{return} | \text{throw} \\
\text{block} & = \text{begin}, \{\text{stmt}\}, \text{end} \\
\text{if} & = \text{"if"}, \text{expr}, \text{"then"}, \text{block}, \text{"else"}, \text{block} \\
\text{while} & = \text{"while"}, \text{expr}, \text{"do"}, \text{block} \\
\text{try-catch} & = \text{"try"}, \text{block}, \{\text{"catch"}, \text{str_literal}, \text{block}\} \\
\text{expr} & = \text{assign} | \text{var} | \text{call} | \text{str_literal} | \text{num_literal} \\
\text{assign} & = \text{expr }\text{"="} \text{expr} \\
\text{call} & = \text{"f("}, \text{str_literal}, \{, \text{expr}\}, \text{")"} \\
\text{return} & = \text{"return"}, \text{expr} \\
\text{throw} & = \text{"throw"}, \text{expr}
\end{align*}
\]

This is briefly description of all ACT nodes.

Statement 'stmt'

Root class for all ACT nodes. Every node is statements.
Code block: 'block'
Block of code. In while-language syntax Nielsen uses concatenation(';') operator to chain multiply statements[5]. However introducing of concatenation operator doesn’t fit well into our clone detection framework. Our engine should handle clones which have few statements inserted or removed from the middle of code. Using concatenation operators results to deep trees, where inserting to the end of block result to change of whole tree. Instead of this we use blocks which contains ordered lists of statements. This flattens tree structure and allows to apply different edit distance algorithms.

If statement 'if'
Conditional statement node. Digest of the Java if-statement. Additionally Java multiple choose conditional statements 'switch-case' are also translated to this node. Our conditional statement always contains 'else' branch even if it is an empty code block.

While cycle 'while'
Simple cycle with precondition. Digest of Java while-cycle. All kinds of Java iteration statements are translated to this node. Always contains body even if it is an empty code block.

Try-catch block 'try-catch'
Digest of Java try-catch block.

Expression 'expr'
Root class for all ACT expressions. Expression is a statement which have value.

Assignment statement 'assign'
Digest of Java '=' expression. According ACT syntax can contain any expression on both sides. We do not divide expression by right and left value in our engine.

Method call
Method call. Has wide variety of uses in our Digest of java dot operator. All Java static method calls, virtual method call and even constructor calls are translated to this node. Moreover this node is a digest of all Java binary operators, such as comparison '==', multiplication '*', addition '+' etc.

Return
Digest of Java return statement

Throw
Digest of Java throw statement.

1.2 Translation ACT
ACT have much less constructs than Java. In some cases we express different Java language statements with same ACT constructs or, in rare cases, we just drop them. Our goal is to
get ACT representation for our similar-code search engine, so we intentionally loose some Java statements, like labels and 'goto' operators. It means that two code fragments which differ only by one label will be considered identical. This precision is enough for our practical purposes, because labels aren’t widely used in modern Java software.

ACT doesn’t have notion of unary or binary operators. So all basic arithmetic, logic and comparison operators are translated to special method call, for example Java expression '3*(2+1)' will be translated to 'f(*,3,f(+,2,1))'.

ACT if-statement is same as in Java except it always have else branch. Java switch-case statement are translated to series of consecutive if-statements. Translator expects that every case block are followed by 'break' statement. If break statement is absent translator would add it artificially, it isn’t semantically correct behaviour but good-enough for our similar code detection.

Java switch statement

\[
\text{switch(a): \{ \\
\text{case 1: \ doA(); break; \\
\text{case 2: \ doB(); break; \\
\text{default: \ doC(); break; \\
\}}}
\]

\[
\text{ACT}
\quad \text{if (f('eq', a, 1) then} \\
\quad \quad \text{begin f('doA') end} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{begin} \\
\quad \quad \quad \quad \text{if (f('eq', b, 1) then} \\
\quad \quad \quad \quad \quad \text{begin f('doB') end} \\
\quad \quad \quad \quad \quad \quad \text{else} \\
\quad \quad \quad \quad \quad \quad \quad \text{begin f('doC') end} \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{end}
\]

Java language has 4 different iteration statements[4]. Whyle-cycle translates as is. Java do-cycle converted to while by repeating the cycle body. For-cycle initialization moves outside of cycle and counter expression are appended to the end of body.

Java while cycle

\[
\text{while(code()) \{ \\
\text{doA(); \\
\}
\]

\[
\text{ACT}
\quad \text{while f('cond') do begin} \\
\quad \quad \text{f('doA')} \\
\quad \quad \text{end}
\]

Java do-while cycle

\[
\text{do \{ \\
\text{doA(); \\
\} while (cond());}
\]

\[
\text{ACT}
\quad \text{doA();} \\
\quad \text{while f('cond') do begin} \\
\quad \quad \text{f('doA')} \\
\quad \quad \text{end}
\]
Added in Java 6 enhanced for-cycle is a syntax sugar for Java SE collection framework API [7]. We replace them with equivalent for-cycle and then translate it to ACT while. For example these two Java code fragment are equivalent in sense of generated bytecode:

Java source code
for (X x : xs){
  do(x);
}

Equivalent Java code
for (Iterator it = xs.iterator();
     it.hasNext();
) {
  X x = (X) it.next();
  do(x);
}

Some statements are completely dropped during translation: different 'goto' statements, assertions and synchronization blocks. 'Finally' blocks isn’t present on ACT, but its content added just after try-catch statement.

1.3 Definition of similarity

How close are two code fragments? Can they be considered as clones? To answer this question we define similarity between two ACT nodes is a float number in an interval from 0 to 1. Similarity should be equal 1 if code is completely identical, and 0 if have nothing in common. Now we can define similarity between every possible ACT nodes.

Definition 1.1 (Similarity function).

\[ \text{Sim}: ACT \times ACT \mapsto [0...1] \]

First and important rule: similarity of two nodes with different types are always 0.
**Definition 1.2** (Similarity for different node types).

\[ Sim(x, y) = 0 \text{ if } \text{type}(x) \neq \text{type}(y) \]

Now we should define our similarity function for every node type starting from numerical literals. We define numerical literals are similar if and only if they are values are equal.

**Definition 1.3** (Similarity for different node types).

\[
Sim(\text{num}(x), \text{num}(y)) = \begin{cases} 
1 & \text{if } x = y \\
0 & \text{otherwise}
\end{cases}
\]

For string literals good candidate is the Levenshtein edit distance[8]. The Levenshtein distance between two words shows the minimum number of single-character insertions, deletions or substitutions required to change one string into the another. Cost of every insertion, addition or substitution is 1. To get value in needed interval \([0...1]\), we normalize output of Levenshtein function by dividing by maximum string length.

**Definition 1.4** (Similarity for string literals).

\[
Sim(s_a, s_b) = 1 - \frac{\text{lev}(s_a, s_b)}{\max(\text{length}(s_a), \text{length}(s_b))}
\]

Using same idea we can define editional distance for two code blocks: edit distance of two code blocks is the minimum number of single statement insertions, deletions or substitutions required to change one code block into other. Insertions and deletions are count as 1 operation, substitutions counted as \(1 - \text{sim}(\text{stm}_a, \text{stm}_b)\) operations. Such approach will help us to handle similar code fragment where only few lines of code were removed or added.

**Definition 1.5** (Similarity for block nodes).

\[
Sim(\text{block}_a, \text{block}_b) = 1 - \frac{\text{lev}(\text{block}_a, \text{block}_b)}{\max(\text{length}(\text{block}_a), \text{length}(\text{block}_b))}
\]

Two method call similarities a computed by same rules as a block similarity, but instead of block statements we compute distance of argument expressions.

**Definition 1.6** (Similarity for method calls).

\[
Sim(f_a, f_b) = 1 - \frac{\text{lev}(f_a, f_b)}{\max(\text{length}(f_a), \text{length}(f_b))}
\]

**Example 1.1** (Block similarity). For example we consider following block ACTs. They differ only by middle function call statement.
Easiest way to convert $\text{ACT}_1$ to $\text{ACT}_2$ is to replace middle statement $f(\text{doB})$ by $f(\text{doD})$.

According block similarity definition this replacements costs
\[
1 - \text{Sim}(f(\text{doB}'), f(\text{doD}')) = 1 - \frac{\text{lev}(\text{doB}', \text{doD}')}{\max(\text{length}(\text{doB}'), \text{length}(\text{doD}'))} = 1 - (1 - \frac{1}{3}) = \frac{1}{3}.
\]
Then similarity between blocks is
\[
\text{Sim}(\text{ACT}_1, \text{ACT}_2) = 1 - \frac{1}{3} = \frac{2}{3}.
\]

**Definition 1.7 (Similarity for conditional statements).** We define similarity of 'if' node as a linear composition of similarities of child nodes.

\[
\text{Sim}(\text{if}(\text{expr}_a, \text{block}_a, \text{then}, \text{block}_a, \text{else}), \text{if}(\text{expr}_b, \text{block}_b, \text{then}, \text{block}_b, \text{else})) =
\]
\[
= C_{\text{if}, 0} + C_{\text{if}, 1} \text{Sim}(\text{expr}_a, \text{expr}_b) + C_{\text{if}, 2} \text{Sim}(\text{block}_a, \text{then}, \text{block}_b, \text{then}) + C_{\text{if}, 3} \text{Sim}(\text{block}_a, \text{else}, \text{block}_b, \text{else})
\]

Constant $C_{\text{if}, 0}, C_{\text{if}, 1}, C_{\text{if}, 2}, C_{\text{if}, 3}$ can be selected empirically such as $C_{\text{if}, 0}, C_{\text{if}, 1}, C_{\text{if}, 2}, C_{\text{if}, 3} >= 0$ and $C_{\text{if}, 0} + C_{\text{if}, 1} + C_{\text{if}, 2} + C_{\text{if}, 3} = 1$ to ensure that similarity always in interval $0..1$. Current version of ACT engine uses following constants $C_{\text{if}, 1} = 0.1, C_{\text{if}, 1} = 0.2, C_{\text{if}, 1} = 0.4, C_{\text{if}, 3} = 0.3$.

**Example 1.2** (Conditional statement similarity). For example if two 'if' statements have identical code blocks, but completely different conditional expressions such as $\text{Sim}(\text{e}_a, \text{e}_b) = 0$ their similarity can be computed as following:

\[
\text{Sim}(\text{if}(\text{e}, \text{b}, \text{then}, \text{b}, \text{else}), \text{if}(\text{e}, \text{b}, \text{then}, \text{b}, \text{else})) =
\]
\[
= C_{\text{if}, 0} + C_{\text{if}, 1} \text{Sim}(\text{e}_a, \text{e}_b) + C_{\text{if}, 2} \text{Sim}(\text{b}, \text{b}, \text{then}, \text{then}) + C_{\text{if}, 3} \text{Sim}(\text{b}, \text{b}, \text{else}, \text{else})
\]
\[
= 0.1 + 0.2 \text{Sim}(\text{e}_a, \text{e}_b) + 0.4 \text{Sim}(\text{b}, \text{b}, \text{then}, \text{then}) + 0.3 \text{Sim}(\text{b}, \text{b}, \text{else}, \text{else})
\]
\[
= 0.1 + 0 + 0.4 + 0.3
\]
\[
= 0.8
\]

Same approach we use to define similarity for remaining node types.
**Definition 1.8 (Similarity for while statement).**

\[
Sim(\text{while}(expr_a, block_a), \text{while}(expr_b, block_b)) = \\
\quad = C_{\text{while}, 0} \\
\quad + C_{\text{while}, 1} Sim(expr_a, expr_b) \\
\quad + C_{\text{while}, 2} Sim(block_a, block_b) \\
C_{\text{while}, 0} \geq 0, C_{\text{while}, 1} \geq 0, C_{\text{while}, 2} \geq 0 \\
C_{\text{while}, 0} + C_{\text{while}, 1} + C_{\text{while}, 2} = 0
\]

**Definition 1.9 (Similarity for try-catch statement).**

\[
Sim(\text{try}(\text{try}_a, \text{var}_a, \text{catch}_a), \text{try}(\text{try}_b, \text{var}_b, \text{catch}_b)) = \\
\quad = C_{\text{try}, 0} \\
\quad + C_{\text{try}, 1} Sim(\text{try}_a, \text{try}_b) \\
\quad + C_{\text{try}, 2} Sim(\text{var}_a, \text{var}_b) \\
\quad + C_{\text{try}, 3} Sim(\text{catch}_a, \text{catch}_b) \\
C_{\text{try}, 0} \geq 0, C_{\text{try}, 1} \geq 0, C_{\text{try}, 2} \geq 0, C_{\text{try}, 3} \geq 0 \\
C_{\text{try}, 0} + C_{\text{try}, 1} + C_{\text{try}, 2} + C_{\text{try}, 3} = 0
\]

**Definition 1.10 (Similarity for assign statement).**

\[
Sim(\text{assign}(lvalue_a, rvalue_a), \text{assign}(lvalue_b, rvalue_b)) = \\
\quad = C_{\text{assign}, 0} \\
\quad + C_{\text{assign}, 1} Sim(lvalue_a, lvalue_b) \\
\quad + C_{\text{assign}, 2} Sim(rvalue_a, rvalue_b) \\
C_{\text{assign}, 0} \geq 0, C_{\text{assign}, 1} \geq 0, C_{\text{assign}, 2} \geq 0 \\
C_{\text{assign}, 0} + C_{\text{assign}, 1} + C_{\text{assign}, 2} = 0
\]

**Definition 1.11 (Similarity for return statement).**

\[
Sim(\text{return}(expr_a), \text{assign}(expr_b)) = \\
\quad = C_{\text{return}, 0} \\
\quad + C_{\text{return}, 1} Sim(expr_a, expr_b) \\
C_{\text{return}, 0} \geq 0, C_{\text{return}, 1} \geq 0 \\
C_{\text{return}, 0} + C_{\text{return}, 1} = 0
\]
**Definition 1.12** (Similarity for throw statement).

\[
\text{Sim}(\text{throw}(\text{expr}_a), \text{assign(throw}_b)) = \\
= C_{\text{throw},0} \\
+ C_{\text{throw},1} \text{Sim}(\text{throw}_a, \text{throw}_b) \\
C_{\text{throw},0} \geq 0, C_{\text{throw},1} \geq 0 \\
C_{\text{throw},0} + C_{\text{throw},1} = 0
\]

## 2 ACT search engine

One to one comparisons of all parsed ACT nodes required quadratic time and isn't possible for large code base. In order to make similar code search fast we introduce the ACT search engine. Engine stores methods ACTs in database with reference to original files and artifacts. For every ACT are stores one or more fingerprints. Fingerprint is a digest from ACT composed in such way that similar ACTs can have equals fingerprints.

Search routine for some specific Java method consist following steps:

1. Parse method to Java AST.
2. Transform AST to our ACT.
3. Compute fingerprints from ACT.
4. Search for all matching ACTs.
5. Compute all similarities with matched ACTs.
6. If some similarity is greater than our threshold value, then we found similar code fragment.

Before computing ACT we perform few preprocessing step. We normalize ACT structure by merging and flating block statements. Some statements are removed at this step, for example logging statements. To detect logging we just have base of known logging frameworks such as SL4J, java.logging, commons-logging etc.

Fingerprint does not contain information about variable names. To properly handle the class types we need to have information about full project class hierarchy. It means that we should parse not only whole project but all project dependencies. Moreover similar classes in different project can have different names, so considering class types would significantly limit variety of similar code. For this reason we throw away most of type information in our search engine. There is an exception for basic JDK classes, such as java.lang.String, java.util.Date, primitive types etc.

In order to extend the search scope we associate with ACT not only whole tree but also fingerprint of its biggest subtrees. This technique allows to find similar trees where few statements were added or removed. In this case, you can search for the code not only with the exact structure, but also blocks where one operator has been added or removed.
3 Conclusion and Future Work

During this work we developed framework for finding similar code in Java project. We managed to give the formal definition of a similar code fragments using intermediate code representation ACT. Developed during this work translator allows to convert arbitrary Java 1.7 code into ACT. As ACT is a language agnostic representation this is possible to implement support for another languages.

In future work I plan is to parse multiple open source project from Maven central. Final goal is to implement extension for some IDE(Integrated Development Environment) which would find similar code fragments from maven in reasonable time.
References


Test automation for web applications

Ülari Laurson
University of Tartu

Abstract. There are plenty automated testing tools on the Internet. Finding the right tool to test your program or code is not an easy task. The assessment gives an overview of testing tools and explains how each tool works with code examples. Evaluation points out the differences between tools. Evaluation saves our time that otherwise we would spend on evaluating these tools by ourselves.

1 Introduction

Using the right tool to test a program is really important because not all tools are equally suitable for every piece of code and using the wrong tool may cost you money, time, and reputation. This may happen because the used tool did not find all the bugs or, at least, the most crucial ones. A tool may fail since it is not intended to test that part of the code or the tool is incomplete.

The goal of this article is evaluate automated testing tools. People who are interested in software testing can read this article before using the tools themselves and get some knowledge about these evaluated tools. Based on knowledge gained from reading this article, people can say if one of these tools is right or she has to find some other tool.

In this article, we are going to evaluate four tools; these tools are Selenium, Robot Framework, HtmlUnit, and iMacros. These tools are used to automatically test web applications. Only iMacros is a commercial tool, three other tools are open source tools. We selected one commercial tool, to see if the commercial tool has some advantages or disadvantages over open source tools. We picked testing tools because there are many web applications in the web and each of them needs evaluation. Using automated tools to verify web application is a good idea because it will give feedback or report of tested application. Feedback such as one or several functions do not work, or it behaves differently than it supposed. Using this report programmer can find easily the bug location.

We created evaluation criteria to evaluate each tool. Assessing ease of use (installation, support availability), documentation and using test cases to get a better sense how each tool works and what functionality they have. These criteria give an overview of each tool and highlights of each tool its strengths and weaknesses.

This article is divided into three parts (chapters). The first part describes the evaluation criteria and how each criterion is carried out. The second part of the article will give an overview of all tools and its evaluation. Each tool has its
own subsection. The third part gives a brief summary of the evaluation. In that section, we also point out each tool’s strengths and weaknesses.

2 Evaluation criteria

In this section, we can read each criterion description and how the assessment will be carried out. Each criterion is explained in detail. We will grade each criterion. The grade scale is from one to five. One means that there are important core parts missing and five means everything is like it should be.

2.1 Ease of use

We assessment for how complicated is to get working a tool and how easy is to use the tool to test your program or code. How easy is to find help in tool home page. How understandable is the installation guide and how long takes to set up the tool.

2.2 Documentation

Evaluated documentation availability, structure, availability of assisting materials and examples. Has documentation an introduction guide and a user guide. How helpful are examples and how many examples are in the documentation.

2.3 Test cases

In this section, we will describe tests that we will carry out with each evaluated tool. These test cases must give an overview of the functionality. What functions each tool has and how simple is to use these functions. Does a tool miss any important functions that prevent test case to carry out.

We are trying with each tool to fill a form [1]. There are different form items that must be filled. The questions that require filling text box, multiple choice, checkboxes, date, time, etc. Using these tools, we must fill all the questions in the form. We choose form filling because there are many different tasks that tool has to perform such as clicking, fill text field, select checkbox values, select element in the drop-down list. In this section, we can find code snippets and their descriptions. These snippets should give the reader some intuition, how each tool works.

3 Tools

3.1 Selenium [3]

Selenium is open-source software testing framework for web application. Originally was developed by Jason Huggins in 2004 at ThoughtWorks. Selenium IDE
provides a record/playback tool for authoring tests and you even do not need to know any test scripting languages. For more advanced user there is Selenium WebDriver that allows you to write test cases. In Selenium you can write tests in C#, Groovy, Java, Perl, PHP, Python and Ruby. These written tests can be used to run against most modern web browsers.

**Ease of use**
There are two different part of Selenium that you can download Selenium WebDriver and Selenium IDE. We used WebDriver because it allows us to do more complex test cases. In tool home page the documentation section includes installation guides. Selenium can be downloaded into IntelliJ or Eclipse project with maven script. Maven script can be found in the tool home page. We imported Selenium into IntelliJ because we have more experienced with the IntelliJ.

It took us to install Selenium into our computer about 1 hour. We will give four points for ease of use because in download section an installation guide was missing and did no have any links to installation guide.

**Documentation**
Documentation is easy to find in Selenium home page. Documentation is divided into section and sections differ from each other noticeably. Different Selenium tools are in a different section that makes the documentation very readable.

In documentation are very detailed descriptions of installation, user guides and enough examples to get some intuitions how Selenium works and what can be done with it. We like that we can see examples in multiple programming languages. These examples are in Java, C Sharp, Python, Ruby, PHP, Perl, and JavaScript. In Selenium IDE section all GUI elements are pointed out and described in detail.

For documentation, Selenium gets five points. It documentation has examples in different languages, installation guide, screenshots and easy to follow structure.

**Test cases**
Getting started with Selenium is not very hard. In the documentation, there are several examples that help newbies. You can easily and quickly set up an automation script to test your web application. To get more out of Selenium we use it with some testing framework such as JUnit. Using Selenium to perform some task, such as click buttons, visit links, fill text areas, etc., in your web application and using JUnit to test if performed task worked or it did not work as supposed. We used Selenium and JUnit together.

As we mentioned before Selenium allows you to create tests in multiple languages. We decide to write test cases in Java. Filling out the form took some time because we did not want to use elements IDs because that would make implementation easier and we would not use all functionality that Selenium has to offer, even then we did not use all Selenium functionality.
We will now explain, using code snippets, how we can automatically fill out a form. Whole code and all needed classes can be found in [2].

Selenium code snippet

```java
WebDriver driver = new FirefoxDriver();
driver.get("http://goo.gl/forms/JIx1aaCwDs");

WebElement textBox =
    driver.findElement(By.cssSelector("div.ss-form-entry input"));
textBox.sendKeys("Selenium");

Select dropdown = new Select(driver.findElement(By.tagName("select")));
dropdown.selectByVisibleText("Yellow");
WebElement selectedValue = dropdown.getFirstSelectedOption();
Assert.assertEquals("Yellow", selectedValue.getText());

WebElement submitButton =
    driver.findElement(By.cssSelector("input[type='submit']"));
submitButton.submit();
Assert.assertEquals("Taname!", driver.getTitle());

driver.quit();
```

In above code snippet in the first line, we define that we want to use Firefox browser. Function `FirefoxDriver` refers to browsers and opens Firefox browser. Selenium also supports IE, Chrome and Opera browsers. In line two, we define site where we want to go and in opened browser this site opens. In lines four, seven and 12 we locate an HTML element. First, we must define how we want to find an element. `By.cssSelector` will locate the element by CSS class selector and `By.tagName` is searching element with given tag name, in code snippet we see that we want to find tag name "select". There are more ways to find element such as by ID, class name, name, link text, XPath, etc. Function `findElement` then finds specified element.

In line four, Selenium searches text area and function `sendKeys` will fill the found text area with given text. In this example, we fill text area with string "Selenium". In line eight, `selectByVisibleText` will choose given value in the drop-down list, in this snippet, our chosen value is "Yellow". Using function `getFirstSelectedOption`, we get the element that is selected in the drop-down list and with function `getText`, we get selected element value.

In line 12, we search a submit button and in next line function `submit` will submit the form. We could use `click` function to submit the form as well. `click` automatically clicks on given element. `getTitle` gets page title. In the last line, `quit` will simply close the browser.

Selenium gets four points in test cases. Four points because we need to use Selenium with some other testing libraries.
3.2 Robot Framework [4]

Robot Framework is an open source generic test automation framework. Pekka Kläärck shaped the basic ideas for the Robot Framework in his master’s thesis in 2005. The same year the first version was developed at Nokia Siemens Networks. It is used for acceptance testing and acceptance test-driven development (ATDD). You can write test cases using a testing methodology that is in a tabular format. We can write these tables using HTML, tab-separated values (TSV), plain text, or eStructuredText (reST) format [5]. Users can write and use their own test libraries. These libraries can implement using Python or Java. Robot Framework allows users to write new higher-level keywords from existing ones.

Ease of use
In Robot Framework home page there is no documentation, but there is a link to documentation. Using installation guide is very easy to download and install the tool into your computer. There are several ways to install Robot Framework such as using Python installing package pip, install from source code or use windows installer. Besides working with Python, it also works with Jython, IronPython, and PyPy. We downloaded the tool using Python pip because it is the easiest way. If you have already Python and pip installed into your computer then installing Robot Framework will take less than a minute. If you are using Windows then you have to add directories to PATH. Luckily there is guide how to do it in the installation guide.

Finding help at the tool home page is simple because the home page is divided into sections and there are plenty of useful links if you get into trouble with Robot Framework.

Robot Framework will get four points for ease of use because the installation guide is perfect and its installation takes almost no time, but understanding how to do the test in tables will take some time.

Documentation
Finding documentation at Robot Framework home page is easy because there is a separate section for documentation. There you can find an installation guide, user guide, quick start guide, demos, and videos. As we mention in the last section that installation guide is very helpful. There is a guide for every single step, what we need to take to install Robot Framework into your computer. The user guide is very long and there are different examples.

In the documentation section, there are helpful demos such as demonstrate how to create tests, higher level keywords, and libraries. Videos section we will

<table>
<thead>
<tr>
<th>Points</th>
<th>Ease of use</th>
<th>Documentation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Summary table
see videos that show us, how to combine Robot Framework and Selenium to generate a test for automatic browser testing.

Robot Framework has done a good job in the documentation. Documentation has videos, guides, demos, etc. For this, it will earn five points.

**Test cases**

In order to automatically fill the form, we must download Selenium2Library library because this library allows us to control a web browser. It can be downloaded using Python installation package pip. It was hard to implement test cases because tests are constructed in test case tables and keywords, and it is different from regular programming languages such as Java and Python. As we mentioned in Robot Framework we can write test cases using HTML, TSV, plain text and reST format. We choose to write test in plain text because it is easy to edit and the majority of examples are in plain text.

Below we can see code snippet of Robot Framework test cases and our description. Using this code we can automatically fill the form. In [2] can be found the whole code.

---

**Robot Framework code snippet**

```robot
*** Settings ***
Library Selenium2Library
Suite Setup Go to homepage
Suite Teardown Close All Browsers

*** Variables ***
${HOMEPAGE} http://goo.gl/forms/JIxlaaCwDs
${BROWSER} Firefox

*** Test Cases ***
Fill form
Write text to text area My family is big!
Select from dropdownlist entry_1942833894 Blue
Click submit button and verify that new page opens

*** Keywords ***
Write text to text area
[Arguments] ${text}
Input Text tag=textarea ${text}

Select from dropdownlist
[Arguments] ${dropdownListId} ${value}
Select From List xpath=//select[@id='${dropdownListId}'] ${value}
List Selection Should Be xpath=//select[@id='${dropdownListId}'] ${value}

Click submit button and verify that new page opens
    Click Button name=submit
```
Go to homepage
    Open Browser ${HOMEPAGE}  ${BROWSER}

Above we see test case is divided into four parts or tables: Settings, Variable, Test Cases, Keywords. In the "Settings" section, are defined all necessary libraries, what must be executed before or after executing tests. In this code snippet, we need to use Library settings to import "Selenium2Library" into our test case. Line three, Suite Setup setting means that we call Go to homepage keyword before executing test cases. Suite Teardown setting is used when we want to execute keywords after testing. For this example, keyword Close All Browsers closes all browser windows that test opened.

In "Variables" table are defined arguments that can be used in test case tables and keyword tables. Robot Framework has three types of variables scalars, lists, and dictionaries and their syntax is ${SCALAR}, {LIST} and &{DICT}. This code snippet has two scalar variables, HOMEAGE, and BROWSER. In HOMEAGE variable is stored the form URL and BROWSER we define browser name we want to use.

In "Test Cases" section we can define tests. Right now we have only one test named Fill form. Test cases are constructed from keywords and their arguments. This test case uses three keywords. In line 13, we use user-defined keyword Select from dropdownlist and it has two arguments entry.1942833894 and Blue. We can also use library keywords that can be imported from test libraries.

In "Keywords" table are all user-defined keywords. Right now we have four keywords, Write text to text area writes text to the text area, Select from dropdownlist selects given value from given drop-down list, Click submit button and verify that new page opens clicks on the button, Go to homepage opens given browser with given URL. Let's explain one user-defined keyword in more detail. Select from dropdownlist has two arguments, [Arguments] is used to specify arguments, named dropdownListId and value. Next, we use Select From List keyword to select given value in the drop-down list. In order to do that, we first find drop-down list element using XPath and then we select the right value. List Selection Should Be just controls if we selected the right value.

In Robot Framework, there is possible to tag test cases. Tags can be later seen in test reports and logs. Reports and logs are produced every time test cases are executed. In the reports we can find statistics about test cases such as how many time some test cases are passed or failed, this is collected based on tags. These tags then can be used to include or exclude test cases to be executed. Command pybot --exclude Fill_form RobotFrameworkTest.txt can be used to exclude test Fill_form. With tags, we can specify test cases that are considered critical. We just have to write --critical or --noncritical and the tag name.
3.3 HtmlUnit [6]

HtmlUnit is an open source headless browser for Java programs, it means HtmlUnit is a web browser without a graphical user interface. The first version was released in 2002. Using this tool you can do the same things as in a regular browser, click links, click buttons, fill out forms, visit pages, etc. Mainly the tool is used for testing or to get information from websites. HtmlUnit also works with complex AJAX libraries. Using these complex libraries we can simulate web browsers such as Firefox, Chrome or Internet Explorer. The tool is not a generic unit testing framework. HtmlUnit is not intended to use separately but within another testing framework. Testing frameworks such as JUnit or TestNG. The tool is used to simulate a browser and testing framework tests if the simulation gave us right result.

Ease of use
In HtmlUnit home page there is no proper installation guide because of that it is a little bit hard getting work this tool. In the download section you can download a .zip file where are .jar files. Those files you need to import to your project. Some searching in the website we found dependency information. It has code snippets for maven, Buildr, Ant, Groovy Grape, etc. We made a maven project in the IntelliJ and copied maven dependency code snippet to the .xml file and the rest did IntelliJ.

It takes some time to find proper help on the website, but if you find what dependency you need then it is very easy. It took about half an hour to get working HtmlUnit. If you get stuck using the tool then you are in the tough situation because there are not enough helpful guides in the site. Still there are two mailing list where you can subscribe and post a question if you have some question regarding this tool. For this HtmlUnit will get four points.

Documentation
As we already mentioned there is no big documentation in HtmlUnit home page. We cannot say there is no documentation in the website. In this tiny documentation is described necessary dependencies and brief tool description. In the home page, you can find examples of HtmlUnit usage. These examples are very helpful and will give you first intuition how HtmlUnit works and what can be done with it.

Compared to the other tools HtmlUnit documentation is very simple and has not as many examples as other tools. For this, it receives three points.
Test cases
Implementing test cases using HtmlUnit is hard in the beginning because there are not enough examples in the home page. To get best out of this tool, it is a good idea to use it within another testing framework such as JUnit or some other testing framework. This is even pointed out in the tool index page that is intended to be used within another testing framework. We used JUnit for his examples. Using HtmlUnit, you can click buttons, fill forms, get table elements, get pages, etc.

In HtmlUnit, you can only simulate a browser using Java language. We used the tool to fill a form that is described in evaluation criteria. We made code that automatically fills forms and below you can see code snippet of this code. The complete code can be found in address [2].

```java
WebClient webClient = new WebClient(BrowserVersion.FIREFOX_38);
HtmlPage page1 = webClient.getPage("http://goo.gl/forms/JIxlaaCwDs");
HtmlForm form = page1.getForms().get(0);
HtmlTextArea textArea = form.getFirstByXPath("//textarea");
textArea.setText("My family is big!");
HtmlSelect dropdown = form.getSelectByName("entry.1942833894");
HtmlOption option = dropdown.getOptionByValue("Yellow");
dropdown.setSelectedAttribute(option, true);
assertTrue(option.isSelected());
HtmlSubmitInput button = form.getInputByName("submit");
HtmlPage page2 = button.click();
assertEquals("Taname!", page2.getTitleText());
```

In the first line of above code snippet, we specify what browser we want to use to carry out our simulation and tests. In this example, we use Firefox, but it is possible to use Chrome or Internet Explorer. The second line, we define the page where we want to go, function `getPage` takes care of that. In the next line, we perform a search to find form in our page and get the first form, function `getForms` is used for this purpose.

In HtmlUnit, we can get different HTML elements in many ways. In the lines five, eight and thirteen, we see two possible ways. Function `getFirstByXPath` and `getSelectedByName` is used to search and get HTML elements. Using the first function you can perform more complex search, in this example we search the first element that has "textarea" tag name, but you can perform more complex search than this example. `getSelectedByName` functions searching element by name, this example we are searching "entry.1942833894".

In the sixth row, we see how to write text into text area using function `setText`. Using Htmlunit you can even select one element in the drop-down list. First, you need to find HTML element where the drop-down list is. Next, we
have to select an option, we want to select, using function `getOptionByValue` and using function `setSelectedAttribute`, select the selected option. Described process can be seen in line nine and ten. `isSelected` just returns true if the current value is selected otherwise false.

Performing a mouse click first, we need to find a button element. Next, we just use `click` function and button is clicked. In this example clicking a button means submitting a form. You can also get the next page that opens after the button is clicked and to further simulation and testing with this page. This can be seen in line 14. `getTitleText` gets next page page title.

`HtmlUnit` as Selenium needs testing library. Implementing test case took longer than other tools for that it will receive three points.

### Summary table

<table>
<thead>
<tr>
<th>Points</th>
<th>Ease of use</th>
<th>Documentation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### 3.4 iMacros [7]

iMacros is a commercial tool for web automation, data extraction, and web testing. The initial version was released in 2001 by iOpus. It was the first macro recorder tool that were built for web browsers and form filling. Similar to web testing and form filler software we can add a record and replay functionality of web application. You can automate tasks of web browsers and have full programmatic control over the browser. It allows you to script tasks including most sophisticated tasks. Using this tool we can automate test across many website technologies such as HTML, Adobe Flash, Adobe Flex, Silverlight and Java applets. It also has a command-line interface and an application programming interface (API). These are used to integrate other programs and automate the most complicated functions. iMacros allows us to easily and automatically web harvesting and web crawling.

#### Ease of use

You can use iMacros for 30-days for free, where you can use all functionalities that the tool provides, we used this trial version. Installing iMacros tool is simple. You just download the installer and execute it and setup opens with a guide, where is detailed explained each step and what you have to do in each step. Download includes an online user manual, where is more detail explanations and examples about iMacros. The installation will take only a couple of minutes. It has a user forum, where you can ask a question and it also has how-to guides. iMacros uses macros and JavaScript to simulate tasks and test, if you are not used macros before, then it is a little bit hard. You can download example script that help you to understand macros.
iMacro has a graphical user interface. It has simple and user-friendly user interface. All the basic functionality is easily found. Functionalities such as play or record a simulation, manage macros and help.

iMacro will get five points for ease of use because it is very easy to use.

Documentation

You can get documentation in several places if you download iMacros. One way is to go to tool home page and in the support section is a link to documentation. There is also link to documentation in the tool itself and there is also a link in the tool folder. The documentation itself is easily readable and divided into sections. There can be found basic examples and even advanced examples. It has plenty of examples to get started. There is an FAQ section if you have some questions. If you still have not found the answer to your question, then you can visit the iMacros forum.

As mentioned above iMacros documentation has examples, FAQ section and easily readable, for this iMacros gets five points.

Test cases

Getting started with iMacros is pretty easy. Using record mode you can record your web application usage and save the recording into a macro file, later you can play this file again. There is possible to write a macro file from the scratch or edit existing macro file. Using iMacros, you can record or imitate many functionalities such as clicking buttons, fill out text areas, open tabs and close tabs, etc.

For simulating form filling, we first selected “Record” and filled out the form ourselves. Next, we stopped recording and saved the current recording as a macro file. Later we added some code ourselves, this file can be found in [2]. Below is tiny code snippet from recorded macro file.

Macros code snippet

```plaintext
1 URL GOTO=http://goo.gl/forms/JIxlaaCwDs
2
3 TAG POS=1 TYPE=INPUT:TEXT ATTR=NAME:entry.832151148 CONTENT=iMacro
4 TAG POS=1 TYPE=TEXTAREA ATTR=NAME:entry.342252419 CONTENT=My family is big!
5
6 TAG POS=3 TYPE=INPUT:RADIO ATTR=NAME:entry.22523304 CONTENT=YES
7 TAG POS=3 TYPE=INPUT:RADIO ATTR=NAME:entry.22523304 EXTRACT=CHECKED
8
9 TAG POS=4 TYPE=INPUT:CHECKBOX ATTR=NAME:entry.131341706 CONTENT=YES
10 TAG POS=4 TYPE=INPUT:CHECKBOX ATTR=NAME:entry.131341706 EXTRACT=CHECKED
11
12 TAG POS=1 TYPE=SELECT ATTR=NAME:entry.1942833894 CONTENT=%Yellow
13 TAG POS=1 TYPE=SELECT ATTR=NAME:entry.1942833894 EXTRACT=TXT
14
15 TAG POS=1 TYPE=INPUT:SUBMIT ATTR=NAME:submit
```
In the first line, we see how the URL is defined. In front of this line we see URL, this means that we are dealing with the web address. GOTO means that we are going to this address. If we use TAG command, then we want to search HTML element by tag name or type. POS command specifies element position we are interested. For example, if we have checkbox, where is several values, and we want to select the third one then we use the POS=3 command. This example can be seen in line six.

We specify with TYPE command a tag name or type. TAG POS=1 TYPE:TEXTAREA example means that we are searching a tag called "textarea", such as in line four. ATTR command is used if it is important that element has given attribute. In line three ATTR=NAME:entry.832151148, we search element that has attribute "name" and its value is "entry.832151148". CONTENT command sets element value. Value can be text, Boolean or name.

Let me explain more detail one or two code lines in the code snippet. In line 12, its searching element from the drop-down list, we see that it is searching element by tag name. Next, we want an element that is in position one, the tag has name "select" and it has "name" attribute with value "entry.1942833894", and selected item name value has to be "Yellow". "%" sign means that tool is searching given value, in our example we are searching the value "Yellow. There are two more possible ways, one is by position or index and second is by attribute name (it has to start $ sign). In line 15, we are searching a button, as you see there is no CONTENT command this means the tool clicks this button rather than change its content. The last line gets page title.

Only tested tools iMacro uses image recognition to find page elements. Image recognition is very helpful to test websites with non-HTML based functions such as Flash or Java applets. iMacro can find images everywhere in the screen even if the move around, it does not rely on element coordinates. The recognition works also with text, buttons or any other element that can be found inside the web browser window. Another functionality that other tools does not have is to automate and test file up- and downloads.

Because iMacros has a graphical user interface, making the test case was very easy and took less time than with other tools. For this, we will give five points.

Summary table

<table>
<thead>
<tr>
<th></th>
<th>Ease of use</th>
<th>Documentation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
4 Evaluation summary

<table>
<thead>
<tr>
<th>Tool</th>
<th>Ease of use</th>
<th>Documentation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selenium</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Robot Framework</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>HtmlUnit</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>iMacros</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Above table we see that all tools are pretty equal. Weakest is HtmlUnit because it does not have a good documentation and will take more time to learn this tool. No tool receive max points because each tool has its own weaknesses.

5 Conclusion

Each evaluated tool is different from each other and has its own strengths and weaknesses. Now the main question is how are these tools different from each other? The main advantage that Selenium has over other evaluated tools is that is very flexible. We can write test cases almost in any programming language. We can combine Selenium with other frameworks. This makes Selenium differ from others. Robot Framework allows us to write test cases in tabular form and these test cases are very similar to natural language. Robot also allows us to tag test cases and these tags can be later used to exclude, include or specify critical test cases other tool does not allow that. Robot Framework is the only tool that generates a test report. It is the tool for people who need to specify critical test cases and want to see nice report after test execution. The main advantage and why someone should use HtmlUnit is that the tool works as a headless browser. This means it is much faster than other tools and give feedback about tests quicker. The test can run in the background without any visual interruption. What makes iMacro special, is the image recognition functionality because of that we can test Flash or Java applets. iMacro also allows us to automate test file up- and downloads. People who do not like programming fit iMacro because it has GUI. Selenium also has GUI, but we did not test it.

This article still has some limitations. In this article, there are described only basic functionality, but each tool has more to offer. Next step is to try out more complex functions and give an overview about these complex functions. So people who have more experience in the testing field have also something interesting to read and learn.

References

1. Test form, http://goo.gl/forms/JIxlaaCwDs
2. Code examples,
   https://www.dropbox.com/sh/wfuxx17mzk5w6w0/AAEGiEDEr34h2SjxN_mlsbye?dl=0
3. Selenium home page,
   http://docs.seleniumhq.org
4. Robot Framework home page,
   http://robotframework.org/
5. Robot Framework User Guide,
   http://robotframework.googlecode.com/hg/doc/userguide/
      RobotFrameworkUserGuide.html
6. HtmlUnit home page,
   http://htmlunit.sourceforge.net/
7. iMacros home page,
   http://imacros.net/
Assessing Aspects of the Model of Learning Computational Thinking

TAUNO PALTS, University of Tartu
MARGUS PEDASTE, University of Tartu

Computational thinking (CT) has gained a lot of attention recent years and is becoming an essential part of the education in a dynamic world when facing the future. There are no models to describe systematic approach to teach CT at school that can be practically used to create lesson scenarios. This paper presents the model of learning CT and assessing methods that can be used based on that model to teach CT in the comprehensive school lessons. An overview of the model of learning CT is given, assessing methods for the Scratch lessons composed and described. Future studies are done to conduct the lessons using the assessing methods described in this paper and to complement the model of learning CT.

Categories and Subject Descriptors: M.1.4.1 [Social and professional topics]: Computing Education—Computational Thinking

Additional Key Words and Phrases: assessing computational thinking, programming, mathematics, comprehensive school

1. INTRODUCTION

Computational thinking (CT) has gained a lot of attention recent years and is becoming an essential part of the education in a dynamic world when facing the future.

"Computational thinking (CT) is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information processing agent" [Wing 2006]

This definition is most commonly used to describe CT as it focuses mainly on three components of CT:

(1) formulating problems and their solutions (based on problem solving activities),
(2) representing solutions (cognitive aspects of CT) and
(3) carrying out the solutions by an information processing agent (explaining the usage of computers, machines or other human beings to solve the problems).

On the basis of those three aspects and a literature review, a revised model of learning CT has been created by Palts and Pedaste (2015) shown on Figure 1.

This model (Figure 1) includes three main layers of CT:

(1) The middle circle includes an interaction between a human and a computer or between a human and another human. This represents the kind of interactions can be used to create lesson scenarios.
(2) Circles forming the middle circular layer of the model consist of the cognitive activities of CT from the conceptual model of CT [Wenchong et al. 2014]. Cognitive activities of CT are are structuralization, formalization, optimization, association and interaction, reuse or sharing. These five elements can be rotated around the center and used dynamically without in any fixed order. That way any of those elements can be included in the lesson scenario as well.
(3) Outer circle consists of problem solving steps from the engineering design process (Massachusetts Department of Education, 2006). Those 8 steps are included in the model to go through the ac-
activities that occur during the process of learning CT. The first step in the process of learning CT should be identifying the problem. The arrow pointing from step 1 to step 2 indicates that after the problem is identified, then research is done, and after that, possible solutions are developed. The steps are following each other. The circular arrows indicate that when one problem is solved (step 8), a new one can be started from the beginning (step 1). From that layer any of those problem solving activities can be included in the lesson scenario, too.

Aim of the paper is to find assessing methods for assessing various aspects of CT according to the model of learning CT.

2. METHOD

Model of Learning CT includes various aspects of CT [Palts and Pedaste 2015]. It is a theoretical model formed on basis of a systematic literature review.

This paper presents ways to assess CT in Scratch projects. One way to assess Scratch projects is to analyze automatically cognitive aspects of CT from the project files [Brennan and Resnick 2012].

ACM Transactions on Applied Perception, Vol. 1, No. 1, Article 1, Publication date: November 2015.
Problem solving activities can not be seen from the project files and have to be assessed either by observing the activities or by doing interviews [Brennan and Resnick 2012].

3. RESULTS
Dr. Scratch is an environment that can be used to analyze Scratch projects. In order to do that, a project must be either downloaded and uploaded to the Dr. Scratch or the public weblink of the Scratch project inserted into Dr. Scratch.
Then Dr. Scratch analyzes the project and gives an overview of the aspects of CT from the projects (Figure 2). Aspects are evaluated from 0 to 3 to indicate level of difficulty of various aspects in the projects.

This way different aspects of CT can be assessed in 4 different levels (from 0 to 3). For example Flow control includes the following improvement possibilities:

To get 1 point: The most basic way to control the behavior of your characters is creating a program composed of a set of blocks are executed one after another, as we see in the picture (Figure 3):

![Fig. 3. Getting 1 point in Dr. Scratch](image)

To get 2 points: Sometimes when we want a set of blocks is constantly repeated, instead of repeating the same blocks over and over again, you can use a repeat instruction allowing achieve the same effect, but more comfortable and manageable way. Consider a couple of examples (Figure 4):

![Fig. 4. Getting 2 points in Dr. Scratch](image)
To get 3 points: Sometimes we don't know previously the number of times you want a set of blocks is executed, as this depends on a given situation. In these cases, the block 'repeat until ...' is really useful. Here's an example (Figure 5):

Comparing to the model of learning CT, Dr Scratch does not assess exactly the same aspects as are described in the model of learning CT, but included into the same categories as follows:

—Flow control - optimization.
—Data representation - standardization.
—Abstraction - formalization.
—User interactivity - association and interaction.
—Synchronization - association and interaction.
—Logic - formalization.

This tool can be used to assess the cognitive aspects of CT found from the Scratch projects. In order to find the problem solving aspects of the model, lesson must be observed.

Scratch projects and observation data can be analyzed to assess the aspects of CT learned in the lesson. Retrieved information can be used to validate the instruments and prepare the lessons for the main study.

4. DISCUSSION
This paper includes the description of the model of developing CT, which can be used to develop CT in comprehensive school. Main goal of the paper is to find assessing methods according to the model of learning CT to teach aspects of CT in a lesson. In order to do that, Dr Scratch is used to assess cognitive aspects of CT and lesson observation and interviews can be used to assess problem solving aspects.

Next step is creating the lesson scenarios (lesson plans, exercises and worksheets) and conducting the lessons according to the scenarios. Lessons should be observed, too. After that the results are analyzed descriptively. Interviews and questionnaires can be used to assess the aspects of CT, which could not be assessed by observation.

This study would be a part of the bigger study to eventually complement the model of learning CT and finding the best practices to develop CT.
REFERENCES
New languages fast as Java

Introduction to the Truffle framework

Tõnis Pool
University of Tartu
tonis.pool@gmail.com

Abstract

This report gives a high level overview of the Truffle Language Implementation Framework developed by Oracle Labs in collaboration with the Institute for System Software at the Johannes Kepler University. Truffle aims at making implementing new programming languages that have good performance much easier. The main approaches are to reuse existing JVM services and to use AST node rewriting with partial evaluation of the AST interpreter. These, together with the new Graal just-in-time (JIT) compiler, give the necessary performance gains.

1. Introduction

Programming languages are tools for solving problems with computation. As such there must be different languages for different problems and new languages for new problems or insights. But implementing programming languages is hard. Making their performance acceptable even harder.

Often novel programming languages take many man years to gain good performance or die off before they ever get there. Gaining good performance usually means implementing complicated runtime systems such as compilers or garbage collection, that are tricky to get right.

Truffle is a framework implemented in Java that aims to help language implementers by reusing existing runtime systems. Language authors only need to implement an interpreter in Java for their new language. With the Truffle system they could already have performance almost in the same order of magnitude as Java[1].

The overall architecture is show on Figure 1.

Truffle builds on top of the standard JVM, by making use of its existing runtime services, such as garbage collection. The Truffle Optimiser layer represents partial evaluation using the Graal compiler, which is discussed further in Section 3.

Next comes the Truffle API level, which symbolises the Java API-s Truffle provides to construct the AST Interpreter and perform node rewriting, which is discussed shortly. AST Interpreter is what the language implementer has to implement and the overall goal is to run applications written in the guest language whose runtime is implemented using the Truffle framework.
2. **Self-Optimizing AST Interpreters**

Implementing an AST interpreter for a new language in some existing language is often the simplest way to make the language executable. As languages require a parser to parse the source code, making the parsed tree of symbols executable can be much easier than translating those symbols into machine instructions or another programming language.

The typical way to write an interpreter is to add an `execute` method to each parsed node, which then carries out the actions that the node is meant to do. The downside of interpreters is that it usually amounts to a lot of virtual method calls and the `execute` methods tend to turn into highly polymorphic call sites, which means they are difficult to optimize and slower than they could be.

A way around that is to introduce the concept of node rewriting so that the AST can be changed depending on the runtime values of the program. This is one of the key aspects of the Truffle framework. Nodes in the AST are turned into specialised forms that can perform the operations faster for specific variable values seen during runtime, but they lose the ability to handle all possible cases.

This means that a certain set of assumptions are made that are then checked before calling the specialised nodes. If an assumption fails the specialised node is replaced with a more generic one that could handle more possible cases, but not necessarily all the cases. The process is repeated until the AST achieves a stable state where subsequent rewrites are unlikely, although not prohibited. This naturally means that there must be an upper bound to the number of specialisations and rewrites.

2.1. **Examples of Rewritings**

Depending on the runtime values of operands and variables the AST interpreter could do one of the following optimisations:

1. **Operation Specialization** - In dynamic languages the performed operation can vary greatly depending on the types of operands. The simple “+” operator can mean integer or floating point addition, string concatenation or a method dispatch depending on the language semantics. Thus the `Addition` AST node can be specialised to different forms - one that does simple integer addition, next one that does floating point addition, then string concatenation, then method dispatch and finally generic node that can handle all the possible cases and combinations of cases (adding string + integer etc.).

This specialisation is not limited to dynamic languages. For example, when a programmer has specified an addition on 2 double variables, but in runtime they are both actually integers and the addition does not overflow then doing an integer operation is beneficial as on all current architectures integer operations can be performed faster than floating point computations.

2. **Polymorphic Inline Caches (PIC-s)** - Languages with dynamic dispatch have to lookup the call target of the function call before executing it, which is costly. It has been observed, however, that function call targets change seldomly and can be divided into 3 classes:

(a) **monomorphic** - only one target  
(b) **polymorphic** - a few targets  
(c) **megamorphic** - arbitrarily many targets

Polymorphic Inline Caches is a well known technique for caching and linking up to a small number of call targets, so that when a function is called the cached list is iterated for a type match. AST rewriting can easily encompass PCI-s by creating a chain of cached nodes. For every new entry in the cache, a new node is added to the tree. When the chain reaches a certain predefined length, the
whole chain replaces itself with one node responsible for handling the fully megamorphic case[2].

3. **Dispatch Chains** - Dispatch chains are a generalisation of PIC-s for example used to optimize reflective method invocation. Essentially dispatch chains build layers of caches, where for reflective method invocation the first layer is caching on the method name and the second layer is a classic polymorphic inline cache for the resolved method. Similar technique can be used to optimize reflective field and global access operations.

The AST can be modified with new nodes as new method names are encountered during runtime and new cache nodes added. Dispatch chains can be used to eliminated the overhead of reflective operations[5].

2.2. **Tree Cloning**

One immediate problem that appears when specialising nodes is when different callers call methods with different parameters, potentially causing the method node to revert back to the most generic (slowest) variant. To combat this Truffle uses AST level inlining of hot call sites, where the AST for the called method is copied into the call site AST. Cloning allows Truffle interpreters to gather context sensitive profiling information.

Figure 2 illustrates tree cloning process, where the big circles represent methods, and the small circles represent AST nodes[3].

![Figure 2: Cloning a subtree from a hot code path](image)

Thanks to the use of profiling and AST rewriting the AST interpreter is able to achieve speeds comparable to the speed achievable with language-independent bytecodes[3].

3. **Partial Evaluation**

Node rewriting still leaves the interpreters with the overhead of dynamic dispatch between nodes. This is where partial evaluation comes in to give the necessary performance gain. Partial evaluation is the process of taking a computation \( \pi \) with \( m + n \) variables \( c_1, \ldots, c_m, r_1, \ldots, r_n \) and substituting known values of \( c_1, \ldots, c_m \) into \( \pi \) resulting in a computation with \( n \) variables \( r_1, \ldots, r_n \)[6]. Meaning we substitute the known variables where it is possible and leave unknown variables into the computation.

Truffle leverages this to Just-in-time (JIT) compile parts of the AST tree. They count the number of invocations of a (sub)treenode and reset the counter in the event of a node replacement. When the number of invocations on a stable tree exceeds a threshold, Truffle assume that every node of the tree is constant and therefore many values in the tree can also be considered as constants. The tree is compiled into machine code with dynamic dispatch turned into direct calls, thus removing one of the main performance penalties of interpreters[2].

Partial evaluation and compilation is performed by the Graal compiler, which integrates with the Truffle framework, but will not be examined in much detail. The result of a partial evaluation is represented in a special Graal high level intermediate representation (IR) which then goes through further optimisations, such as method inlining, constant folding etc.

One problem yet to be solved is when speculative specialisations turn out to be wrong after AST has been partially evaluated and turned into machine instructions. Truffle handles this in combination with Graal by turning any code that would modify the AST into a de-optimisation call after JIT compilation that returns the program execution to the interpreter.
Listing 1: Hypothetical addition node using Truffle DSL

```java
@NodeInfo(shortName = "+")
@NodeChildren({@NodeChild("leftNode"), @NodeChild("rightNode")})
public abstract class AddNode extends ExpressionNode {

    public AddNode(SourceSection src) {
        super(src);
    }

    @Specialization(rewriteOn = ArithmeticException.class)
    protected long add(long left, long right) {
        return ExactMath.addExact(left, right);
    }

    @Specialization
    protected BigInteger add(BigInteger left, BigInteger right) {
        return left.add(right);
    }

    @Specialization(guards = "isString(left, right)")
    protected String add(Object left, Object right) {
        return left.toString() + right.toString();
    }

    protected boolean isString(Object a, Object b) {
        return a instanceof String || b instanceof String;
    }
}
```

with the correct runtime values. When enough interpreter executions have happened again after a deoptimisation, the process repeats itself with the AST being considered stable once more and compiled.[7]

4. IMPLEMENTING GUEST LANGUAGES

Truffle has been used already to implement runtimes for C, JavaScript, Ruby, Python and R[8]. All of the dynamically typed programming languages have reached at least comparable or better speed than their default runtimes. In addition Truffle has been used to implement a Lisp language called Mumbler, that has tail call optimization, while Java itself still does not[9]. This list of languages alone shows that Truffle can be used to implement a large variety of programming languages successfully.

This might be surprising as the language author is constrained to only the language features Java offers to write the AST interpreter. But Java (or the JVM more broadly) is an extensible runtime that can be used to emulate or retrofit different techniques or language features that are not natively supported. As an example Quasar is a Java library that provides high-performance lightweight threads, Go-like channels and Erlang-like actors[10], while JRebel is a tool that allows you to reload code changes in a running Java application[11]. None of which is possible in the standard Java programming language.

The following sections cover the main tech-
niques how Truffle helps the language implementor to achieve good performance with their new language.

4.1. Truffle DSL

As mentioned in Section 2 node rewriting is one of the main aspects of the Truffle framework, thus it is important to make it easy for the language implementor to describe possible rewritings. Truffle solves this by offering an declarative Java annotations based DSL that can be used to express possible optimisations[12].

As an example let’s look at a hypothetical addition node that can add arbitrary precision integer numbers and strings together, shown in Listing 1. Starting from the top, the @NodeChildren(...) annotation says that this AST node has 2 children called "leftNode" and "rightNode". The first add method takes in two Java long type variables and adds them using ExactMath.addExact which throws an ArithmeticException when the addition overflows the 64 bit long values. In that case the addition node should be rewritten to a more generic one that uses BigInteger data type that has arbitrarily large precision. The @Specialization annotation on the method does just this - it tells the Truffle system to rewrite this node to a "broader" one when that exception is thrown.

Specialisations are guarded by various guards provided by the DSL in the form of attributes on the @Specialization annotation[12]:

1. Type guards - In order for the specialisation to hold the type of the arguments must match those of the method parameters.

2. Method guards - The guards attribute defines the set of methods that need to return true in order for the specialisation to hold. An example of this is the isString(left, right) value in Listing 1.

3. Event guards - The rewriteOn attribute says to trigger a re-specialisation when a specific exception occurs.

4. Assumption guards - The assumptions attribute defines a set of expressions whose return types must be com.oracle.truffle.api.Assumption. When any of the returned Assumption types are invalidated during runtime the node is triggered for re-specialisation.

Besides allowing to declaratively describe node specialisations the truffle DSL provides many different shortcuts such as defining implicit casts between different data types, explicitly controlling type checks for some types and to short-circuit boolean expressions if that is needed in the guest language.

The curious reader might wonder, but how does it all work in practice? The Truffle DSL uses the standard Java annotation processor to generate additional source code during compilation. The whole process is described on Figure 3.

In the end the Truffle DSL helps the language implementer to avoid a lot of boilerplate code. As an example the JavaScript implementation in Truffle in the end of 2014 had ~16800 lines of code, but ~157400 lines of generated code by the annotation processor[12].

Figure 3: Truffle DSL annotation processing pipeline

4.2. Modelling Types

Next let’s look into how the Truffle framework can help the language implementer to model
objects and types. Originally Truffle left this problem in the hands of the implementer, but as it was a common enough problem they started to offer a reusable solution that was called the Truffle Object Storage Model (OSM). The major challenge here of course is how to make the OSM general enough that it would not constrain potential languages, but at the same time performant enough that it would not be the bottleneck.

An object model in general defines the properties of objects in a specific programming language as well as the semantics of operations on them. Truffle OSM provides basic tools for emulating object models of guest languages. Objects in Truffle OSM consists of a variable number of mutable properties, each with a name, a value, and a set of attributes[13].

The implementation of Truffle objects uses two separate parts[13]:

1. **Object storage** - Contains the per-instance data, similar to a Java object in the heap.

2. **Shape** - Defines the overall structure of the object by defining the mapping of member names to locations in the object storage. This is similar to a Java class.

The object storage class defines predefined number of slots (fields on a class) for primitive types and reference types. When an objects needs more than the predefined number of slots it can use an extension array to quietly overflow the data there. The reason to do this is that field access is faster than array access.

As mentioned a shape acts as a class for an object, except unlike Java classes, shapes can transition into new shapes, thus the "class" of an object can change in runtime. Shape instances are immutable and shared between all objects with the same mappings. Shapes form a hierarchical tree. Whenever shape is changed during runtime, say a property was deleted or added, a new shape is created with pointer to its parent (ie. the previous shape).

Setting and accessing properties will be specialised to the specific shape automatically by the Truffle OSM, to avoid the potentially costly operation of finding the storage location for the property. Additionally custom attributes on the properties can be defined to achieve needed custom behaviour, Truffle OSM provide the hidden attribute which indicates that the property isn not visible to guest language[13].

As an example let’s look at the JavaScript code snippet:

```javascript
Listing 2: Creation of a JavaScript object

var course1 = {id: "OOP", eap: 6};
var course2 = {id: "ALG", eap: 6};

course1.room = "404";
```

At first both course1 and course2 start off pointing to the same shape, but as a property is added to course1 a shape transition occurs and a new shape is created. The result looks like Figure 4 where blue boxes correspond to shapes. If later the course2 object would get an room property as well then the objects would share the same shape again.

```javascript
Figure 4: Example shape hierarchy
```

## 5. Alternatives

Truffle is not the only framework that is trying to make implementing new languages easier and the resulting language performant nor is node rewriting with partial evaluation the only means to achieve it. There is a comparable language framework called RPython that similarly to Truffle works by JIT compiling an AST interpreter, but instead of partial evaluation it uses tracing execution of the interpreter to compile a specific path through the code.

Similarly to Truffle RPython has a set of annotations that language implementors can
New languages fast as Java • December 2015

use to give additional information about the language and runtime constraints. As code is compiled based on a specific path through the interpreter RPython also needs to set guards for the conditional control flow it passed. If a guard fails it has to return execution to the interpreter.

A recent comparison found them similar in terms of steady state top performance, with a bit of an edge towards Truffle. Then again as RPython does not require the language implementor to specifically declare the potential AST node specialisation cases the language implementations could require fewer lines of codes compared to the Truffle framework[1].

6. Summary

As seen Truffle provides a comprehensive framework for implementing new programming languages that perform well. The main techniques that enable this are node rewriting together with partial evaluation that is compiled on the fly to native code.

In my opinion the Truffle framework dramatically simplifies the implementation of a language and takes a “batteries included” approach, where the main task for a language architect is to create an executable AST interpreter in Java and the framework takes care of the rest. It should significantly lower the bar for the needed knowledge to create or prototype a new language, whether it is good or bad only time will tell.

References


New languages fast as Java • December 2015


PROGRAMMING LANGUAGES AND ENVIRONMENTS USED FOR TEACHING ROBOTICS IN ESTONIAN SCHOOLS

MARIKA REINMETS
PhD student of Tartu University
Robotics teacher of Tallinn Sydalinna School

1. INTRODUCTION

It is currently very important to cultivate the next generation IT professionals and to develop children and young people interested in the technical disciplines from early age. For this purpose, robotics is included in the schedule of many kindergartens and schools, and also programming is taught to children.

As there are many types of robots which are used for teaching, there exist many programming languages that are used for programming them even though some of the languages can be used for programming different robots.

Figure 1. WeDo, NXT and EV3 robots

Can be argued that in Estonian schools-kindergartens and robotic groups mainly are using the following types of robots:

1. Lego WeDo robots
2. Lego NXT robots
3. Lego EV3 robots

This conclusion is builded by from these three arguments:

A) For what robots robotics teachers of different schools are mostly asking help or are giving advice to each other in Schoolrobotics web (Kooliroobotika list).

B) What kind of robots are carried out in Estonian robotics competitions where the pupils are participate (FLL, FLL Junior, Robomiku, Võru Tsõõr, Technology Days - 5 most popular robotics competitions oriented specially for students.)
C) What kind of methodological training for teachers is organized (by University of Tartu, by Tallinn University of Technology, Vaata Maailma Foundation, Robootika MTÜ, and HITSA).

In addition to these three most popular robots in some schools owned and used to a lesser extent also the following robots:

4. Bee-bots robots  
5. Arduino robots  
6. RaspberryPi robots  
7. Homelab (based on ATmega 2561 microcontroller)

![Figure 2. Bee-bots, Arduino, RaspberryPi and Homelab robots](image)

The following programming languages or programming environments are used for programming the robots:

1. LEGO Education WeDo Software  
2. NXT-G (LEGO MINDSTORMS Edu NXT (NXT x.x Data Loggin ja NXT x.x. Programming))  
3. LEGO MINDSTORMS EV3 Home  
4. LEGO MINDSTORMS Edu EV3 Edition  
5. Scratch (Enchanting)  
6. NXT-C (Not eXactly C)  
7. BricxCC Bricx Command Center)  
8. Python  
9. C, C++

Some of these programming environments are created by robot manufacturers for managing a particular type of robots, but some languages (e.g. Python (https://www.python.org) and Scratch (https://scratch.mit.edu) allow controlling the majority of robots.

Some limits in using programming languages in robotics are put down by the rules on robotic competitions (or by sponsors of these competitions). For example in the FLL League are expected to use Lego official software as Lego Mindstorms Edu NXT or Lego Mindstorms EV3 different editions or NXT-C and are refused to use some freeware (Scratch, Enchanting etc). These limits in competitions are having an effect to the using different programming languages in schools too.
Taking together the information of robots manuals, training materials for teachers and various additions and the personal experience as robotics teacher drawn up the following table where is shown horizontally the most common robots used in Estonian schools and in the vertical axis programming languages which can be programmed these robots. Crosses are marked on each line of these languages, which can be given to program the robot, however these bold cross languages which are for the robot - a so-called default languages for them.

Table 1

<table>
<thead>
<tr>
<th></th>
<th>WeDo</th>
<th>NXT</th>
<th>EV3</th>
<th>Arduino</th>
<th>Raspberry</th>
<th>Homelabor</th>
</tr>
</thead>
<tbody>
<tr>
<td>WeDo</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NXT-G</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NXT-C</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scratch</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, C+</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

My paper is about the various programming languages/programming environments for children, what robots with can be programmed with different languages and what opportunities are provided by these languages/environments for controlling the robots used in kindergartens and schools focussing in this theme only for using programming languages for teaching robotics in Estonian schools and kindergartens and robotic clubs for children in age 5 to 19 years.

2. THE PROGRAMMING LANGUAGES MOSTLY USED IN ESTONIAN SCHOOLS

2.1. Programming "language" used for programming Bee bots robots

The Bee-Bots robots https://www.bee-bot.us/ are designed especially for preschoolers and elementary schools, so there is also the programming instruments used is most simple: Be-bots robots had not separate programming environment in which them can be pre-programming but we can programming them only in holding a certain order button presses, a sequence which starts the robot moves to the required direction.

Bee-bots robots have seven buttons on the back: forward, back, right, left, cancel, pause, and go, and as a result of the combination of them can be directed robot to a certain point of the room. Always these robots are used to teach programming on various topics (food, zoo, alphabet, numbers, etc.) associated with the large study grid but which you have to move according to program robots to perform a specific order and some specific tasks, for example the exercise to pass the letters on the floor-mat in alphabetically order.
2.2. WeDo programming environment


The target users of WeDo robot are particularly students of primary schools, but they are also used in the older groups in kindergartens. WeDo is a graphical programming environment. To use this environment young programmers need a good skill to use a computer mouse, to know the numbers and letters and can already to read.

![Programming environment WeDo](image)

Figure 3: Programming environment WeDo

A variety of activities denote in different colors specific boxes at the images, which the user must use the mouse on the computer screen to pull them in current order for programming. Similar activities are identical color, the little boxes to controlling for electronic components are the same color, but with different images, for example, the boxes for program the motors are green in color, but according to the motor rotation direction with arrows pointing in different directions, separate blocks are a blocks of the engine speed and engine shutdown. Similarly, sensors, eg one orange block directly, the second inclined to the right and the third to the left.

To specify the commands to some of the blocks can download the additional labels with numbers or text, for example, to specify the amount of time the engine is running or standing, to give the tilt sensor right place or angle with additional the parameters or to produce sound during the work of programm or enter the text that appears on the screen.

Like in "real" programming languages, there is also in WeDo a block what indicating periodic activity or cycle, what students often nominated by their peculiar shape as snail or ear. The cycle can be used indefinitely or for a specific
number of times or so long as held a particular activity, such as tilt sensor take
the fixed position or motion sensor detects motion in front of it.

WeDo programming environment is also possible to send messages, for example,
in the course of an activity, the data collected by the sensors triggered upon a
new action for illustration program/robot work can display text or images to
screen, to add sounds, add the "code" to the comments.

WeDo robots have not autonomous separate power source for this the robot's
motors and sensors are powered from the computer through a USB cable.

To run (and complete) WeDo program can be used in a wide variety of keyboard
keys spaced according to of a small programmer wishes, but if the computer
program itself jams, you always have two options to shut the running program:
either the red rectangle on the screen or take the power off from behind the
robot.

The program made in WeDo environment can not be pre-loaded into the robot,
because there is no 'brains' in WeDo robots. The robot works only cable
connected to your computer with USB, you must take it into consideration in the
construction of robots that require more space for mobility (cars, other vehicles,
robotic animals, etc.). In our school in such cases we are using to program by
laptops, which will be holding a child in hands his vehicle or an animal can to
walk along the corridor.

One of the more interesting peculiarity: is not able to save with their desired
name the program created by WeDo programming environment, save can only to
be initially offered by the program to accept the default name (the first file is
program.wedo, the following program1.wedo, etc.) but a file name can be
changed later when you go directly to the directory where it will be automatically
recorded and can to change it name manually.

Since this last activity, however, is generally too complex for the primary school
students, it is reasonable to kids just to clarify that all programs will
automatically be named "program.wedo" and if some program is definitely
needed to be used more in the future (for example, pupils will go to the
competition Jr. FLL), then the child should notify it to the supervisor and he helps
to retrieve the file from your computer, rename it and to save to a memory stick.

There is not any separate recording button in WeDo, all files are recorded
automatically and in next the opening of the programming environment on the
screen we can always see the last program in this environment it is the same as
used in a computer last time. To open some program previously made/saved are
needed to find the icon of the yellow folder image in the top right corner of the
screen, to click on and to select the desired file dialog window.

The "output" in using WeDo robots is the opportunity to participate with them in
robotics competitions like First Lego Junior (Jr. FLL) for children aged 6-10.
2.3. Lego Mindstorms Edu NXT (NXT-G)

Lego Mindstorms Edu NXT is official programming environment what are using for programming Lego Mindstorms NXT robots and is founded by Lego Company. This environment giving to users two ways to programming its robots.

Figure 4. Programming environment Lego Mindstorms Edu NXT

First (more primary) way by buttons based in the robot brick which allow to use some very primitive way programming: collect the resultants of measuring from the sensors, draw to the screen of brick the text or drive the motors. The other and most popular way to programming NXT robots is to use special programming environment - Lego Mindstorms Edu NXT to do this.

Edu NXT can work in computer where user can prepare the program what he want to use in his robot and to download the ready program to the robot can be used as the wire connection (or the bluetooth connection in some new versions of robots).

There are three level to use for this environment: the base level (green circle as an icon) what contain most basically icons of orders, next level (three squares as an icon) in for these who can already use the data gathered by sensors and to use them in logical or in mathematical transactions and the most higher lever is for users building his own base programs contains himself as components from the lower levels (two blue rectangles as an icon).

In basic level there are 7 big square buttons in the left part of screen what student can drive to right and put these in line to create his program. One of these buttons - sensors - give ways to use five different sensors, if users lead the mouse to this there appeared to right five different icons of sensors what can be to use.
For every icons what already had chose to programm and is active at moment (it is meaning that the button has a narrow electric-blue border around him) will appear to the down of screen wide rectangle contains different parameters user can choose and change to programming this active block of programm. For example in the using motor block user can choose and change ports of motor wires, the number of motors using by this block in the one time, the direction of motors, steerings, power of motors, duration to work in seconds, rotations, degrees or choose a unlimited work to motors and decide what the programm will do after to do this block, brake a programm for the moment or the next block in programm line will start at once after this block work is done.

In the next level of users there are six different icons button in left side of screen and every of these gives to the user four to twelve different buttons of possibility to improvement the program. As in the base level so in advanced level to programm the work every block are used in changing different parameters in the down part of screen, but there are more possibilities than in basic level.

In advanced level there is a supplement to using wires in connecting different blocks of programm and it gives to a young programmer more possibilities to use the data gathered measurings by sensors in the blocks of work motors, other sensors or more.

Using Edu NXT environment there is a good quality that in programming we can use components of the different levels in the same programm what give to teachers good chance to teach robotics step by step adding slowly the single components of advanced level to the using components from the base level.

For better understanding from the steps of programm and his ideas appear in process of programming there is good possibility for users to put his own comments to the programming field next the programm and record them with the programm to the same file what giving in the next using or during correction the same programm to take these notes and comments into consideration.

In the right side of screen there are the windows with hints to building and programming for more than forty different simple robots, places to downloads instructions for building some great robotics projects or use NXTLOG, downloads for wallpapers with robot theme or download additional robotic sounds for projects.

Unfortunately the production of Lego NXT robots will be ended in 2016 and there may be (but it is not sure) rise some problems to obtain the additional parts of NXT robots for this reason in future, but we can use NXT robots in the competitions still.

2.4. Lego Mindstorms EV3 different editions (EV3 Home and EV3 Edu)

Lego Mindstorms EV3 Home and Lego Mindstorms Edu EV3 Edition are the programming environments for programming the most last version of Lego robots - Lego Mindstorms EV3. Both of these environments are quite similar, the most difference of these is in the policy of use: EV3 Home user can freely
download from the official website (http://www.lego.com/en-us/mindstorms/downloads) but to use EV3 Edu user must buy the license.

The second difference is in possibilities: Edu version is special for educators and has two different profiles of users: student profile and educator profile what must to mark at once in the process download the programm to users computer. EV3 Home version is invented to home users and there is not different profiles of users. As the sets of robots selling in stores for everyone and by importer direct to educational institutions are contain some different parts of lego blocks so the building instructions in EV3 Home and EV3 Edu versions are a bit of different.

Figure 5. Programming environment Lego Mindstorms EV3 Home Edition

Using in programming robots with EV3 programming environment is not especially different from the NXT environment although they seems in the first eye very difference. Most of the blocks of programming are the same, only the colours of some blocks are different. The buttons based in the left side of screen in NXT are resettled to the bottom part of screen in EV3 and the possiblility of changing parameters in EV3 is now exactly near the blocks of programming.

EV3 is invented some years after NXT so all the motors and sensors of NXT we can use with the EV3 brick (brain), but to use EV3 sensors with NXT brick is quite complicated.

As EV3 is newer there are essentialy more sounds, sensors and images to use than was in NXT. EV3 programming environment let to the user simply to draw his own images to use with program or to record own sounds in the same time of programming process.

Who is better? NXT or EV3? Nobody knows the true: some pupils think that EV3 as the newer, som that the NXT as the simpler so I think it is depending with what kind of robot the user has the longer experience to use.
2.5. Scratch (and its analogs)

Scratch (and its analogs as Enchanting etc) is a freeware and is one of the most popular programming language for teaching programming in primary schools in Estonia but to using this language in the programming Lego NXT and EV3 robots is not quite usual because it demand special change firmware in robots for this before.

![Scratch 1.4 programming environment](image)

Figure 6. Scratch 1.4 programming environment

Scratch is suitable to programming almost all kind of robots are used in schools: WeDo robots, NXT robots, EV3 robots, RaspberryPi robots etc but in the reason named in the last paragraph and for this that in the FLL competitions is refused to use this freeware scratch is not so often used in robotics lessons as it can be.

There is only one kind of robots - RaspberryPi - what are used in Estonian schools and what had Scratch with them as the default programming language and is used too in programming them.

In robotics competitions used scratch in programming first at all in ROBOTEX because in this competitions participants can himself to choose the language of programming they used in their robots, also theoretically scratch can used in Võru Tsõõr and Robomiku competitions too.
Scratch is very simple programming language/environment what can understand all pupils who already can read. Even the pupils of first year can it use kindly in the second semester of their first schoolyear.

Scratch has two versions: one of them need installing program to the computer and the other version is working in the web and so this can used also in the tablets. In robotics is preferred to use scratch on computers or in laptops because most of tablets unfortunately had not the USB port to connect the cable of robots to program them. And if the internet connection is not very good there are preferred the first version because this version is not needed the internet connection all the time of programming as the second of them.

In scratch the screen of computer is divided to different areas (windows): area for program, area for "screen" (to see what happens), area for blocks of programming, area for sprites etc. The blocks of the same category are same in color and the user must to drag the blocks with the mouse in the right order down together to the area of programming.

2.6. Programming robots with C language (NXT-C, C, C+)

C language (and it’s analogs) can use to programming most of robots using in Estonian schools but since using this language needs more specific knowledges and robotics is more popular around the pupils of the elementary and middle classes so using of this language in school robotics is not especially frequently yet, but there is probable exceedingly that for this time when these young robotics of elementary schools who had started with robotics now are already reach to the grammar school they already will use C as the main programming language for programming their robots as the students who are studing robotics in universities in nowadays.

3. CONCLUSION

There are using very different robots with very wide programming possibilities in our schools, but in connections the limits for programming languages in the most popular competitions for pupils, there are using only some of these possibilities in the robotics studies

The other reason is probably this that our universities not prepair the teacher of robotics in this day. Almost all teachers who teach robotics are the specialists of some other subject, robotics is only hobby for them and they had took part of some robotics cources for teachers.

There are only some courses where these who are interested in robotics can study this area. Most of these courses are based in the some concrete model of robot and the default programming environment for this. Only in this schoolyear some training organisations are starting to offer courses where are studing programming lego robots in C or raspberry robots in scratch.
4. SUMMARY

I hope that my paper could give information and a little overview about this what kind of robots and programming language are mostly using in our schools in this period and how are using them.

Robotics and programming are not in the official curriculum of the primary schools in Estonia but as all schools can choose some extra subjects for their own curriculum, many schools have decided to teach some ICT subjects (computer science, informatics, robotics or programming) to their students. In other schools these subjects are taught in children’s activity clubs. In some kindergartens there are often exist clubs of robotics for the 6 year old children where are they are taught some programming as well.

In computer classes, students acquire the first skills in using computers in interactive learning environments. In robotics classes they learn to build, design and program robots and they also attend competitions using their project work.

Robotics is very popular in Estonian schools: every week I recieve e-mails asking if there are available places in our robotics lessons. There were already 72 teams took part in the First Lego League competitions in Estonia in January and there were first time three regional competition to select best teams to the final competition in March.

5. REFERENCES

1. The Bee-Bots robots https://www.bee-bot.us/
7. List of training courses for teachers of robotics by Progetiiger program: http://www.progetiiger.ee/opetajakoolitus
Detection of data-races in Linux device drivers via static analysis

Vootele Rõtov
vootele.rotov@ut.ee

ABSTRACT
This paper gives an overview of static analysis tools used for detecting possible data races in Linux device drivers. Necessary background about the domain of device drivers is given in an attempt to make the paper self-contained.

Keywords
static analysis, Linux device drivers, data race detection

1. INTRODUCTION
Device drivers hide hardware behind a relatively simple interface, offering the operating system a way to make use of the underlying hardware, without the need to handle the device specific complexity.

This task is rather complex, as the operating system requires a lot from the exposing interface, including support for concurrent access.

Considering the fact that the drivers are written in a low-level language, they are very prone to bugs in general and specifically to concurrency bugs, among them data-races.

In this paper an overview of Linux device drivers and the impact of data races in them will be given. The general ideas for finding data races in the device drivers are then introduced.

Lastly, reader is introduced to a number of tools that either detect data races in Linux device drivers or that use the general ideas in some other domains.

2. LINUX DEVICE DRIVERS
2.1 Overview
To explain the role of device drivers, lets introduce two common devices that are usable on computers running a Linux operating system.

First of them is /dev/tty, a device that represents the terminal controlling the current process [20]. Opening a terminal in Linux and writing to the device results in the text being displayed in the terminal. One can try this with the following command:

`echo "Hello" > /dev/tty`

Second device we will introduce is /dev/null, a device that discards any data sent to it. So, redirecting text to this device has no effect. One can send text to the device as follows:

`echo "Hello" > /dev/null`

Lets note that although the two devices act differently, they both can be used the same way – we can redirect text to them (and redirect text from them). This allows us to group the two devices together when reasoning about how to use them. The benefits of this are far greater when the number of devices that can be grouped together grows.

The role of Linux device drivers is to enable such groupings that can contain arbitrary amount of devices.

This is done by enabling access to the devices via a small set of well-defined interfaces that other parts of Linux kernel can make use of.

In this paper we will focus on a subset of device drivers called character (char) device drivers. Character device drivers allow unbuffered access to the underlying ‘hardware’. Both of the drivers for the example devices are char drivers.

2.2 Interface of a character device driver
Devices are exposed to end user as entries in the file system and so it is natural to use the same terminology for talking about device drivers operations as about normal file operations. So device drivers expose end points for reading, writing to, opening and releasing the device. This list is not exhaustive[20], include/linux/fs.h, but it is not necessary for a driver to expose them all.

As a running example we will introduce the driver Counter. Counter keeps track of the difference between how many times kernel has read from it and how many times kernel has written to it.
In the following code snippet, the parameters of the functions are omitted as they are not relevant for this example.

The `read` function of `Counter` increases the global counter by one:

```c
static ssize_t file_read(...)
{    
  ++i;
  printk("++i, new value: %d \n", i);
  return 1;
}
```

The `write` function of `Counter` decreases the global counter by one:

```c
static ssize_t file_write(...)
{    
  --i;
  printk("--i, new value: %d \n", i);
  return s;
}
```

And both the `open` and `release` functions output info to kernel log.

```c
static int file_open(...)
{    
  printk("Opening device \n");
  return 0;
}
```

```c
static int file_release(...){
  printk("Closing device \n");
  return 0;
}
```

When a user reads from a file that exposes the driver, one can see that the file was opened, read and then closed. For a concrete example,

```
head -c 1 < counter
```

produces the following entries in the kernel log:

```
vootele kernel: Opening file
vootele kernel: Increasing i, new value: 1
vootele kernel: Closing file
```

And similarly, when writing to the file, for example with

```
echo hi > my-driver
```

then file is opened, written into and then closed.

Drivers expose the device to the kernel via `file operations` structure. When registering a driver to the kernel, the `file operations` structure is registered with kernel and kernel can invoke any of the methods specified in the structure.

The `file operations` structure of `Counter` is

```c
static struct file_operations f_ops = {
  .write  = file_write,
  .read   = file_read,
  .open   = file_open,
  .release = file_close,
};
```

Device drivers also expose `init` and `exit`, which kernel can use to register and de-register the device driver. In case of `Character`, registering the device initializes the counter `i` to zero and then registers the `file operations`.

### 2.3 Data-races in device drivers

When registering our driver, we exposed multiple endpoints to the kernel. From that moment onwards, kernel can invoke any of the registered methods, at any time, until the driver is de-registered.

The multitude of entry points to the programs and lack of control over when they are entered makes avoiding data races a challenge when writing Linux device drivers.

For an example of a data-race in device drivers, it is quite obvious that when one writes to a ‘device’ exposed by the example driver concurrently, then the value of counter after all the write operations is not determined – it is easily possible that between the read and write operations of the counter, its value gets updated.

To avoid data-races, Linux kernel offers multiple locking primitives. One of the most commonly used locking primitives is the mutually exclusive lock (mutex). It also the one that one should use, if possible [20, Documentation/locking/mutex-design.txt].

Another popular locking primitive in `spinlock`, that blocks until the lock has been acquired (same behaviour has been added to mutex). Its name comes from the tight loop that is used to acquire the lock. Spinlocks also offer a lock that differentiates between reads and writes, allowing multiple concurrent reads.

To eliminate a possible data race from our example driver, we could add a lock to the driver and acquire it on every read and write operation. With the added lock, the read operation would look as follows:

```c
static ssize_t file_read(struct file *fp,
  char __user *buf, size_t buf_size,loff_t *pointer){
  mutex_lock(&my_mutex);
  ++i;
  printk("++i, new value: %d \n", i);
  return 1;
  mutex_unlock(&my_mutex);
}
```

where `my_mutex` is a static mutex.

After adding the same locking pattern to write operations, data-races will no longer plague the example driver.

There are also alternatives to straightforward locking that one can use when writing Linux device drivers to eliminate
data races, such as `seglocks` and `read-copy-update`. These fall outside the scope of this paper, as they are rarely used compared to traditional locking and as of now, static analysis tools do not offer much support for them.

### 2.4 Impact of data races in device drivers

We have seen that data races are a real possibility in device drivers and also that there are ways to protect oneself against them.

As previously discussed, Linux device drivers usually have multiple entry points and no control over when they are entered. Also, the device drivers are written in C, a low level language, that makes reasoning about them quite challenging. Furthermore, quite often the person writing a driver is an expert on the device that the driver is for, and not so much an expert on Linux device drivers themselves. This makes correct usage of constructs that help to avoid data races difficult and error-prone.

Empirical studies validate this assertion. In a study done by Chou et al. [3] it was found that error rates in device drivers are three to seven times higher than in rest of the kernel. The situation seems to have become better over time. In the follow-up study conducted by Palix et al. [13], the error rate in device drivers improved, but drivers still contain the highest number of errors.

In a study done by Ryzhyk and others [16], out of the 498 bugs found between 2002 and 2008 in 13 selected drivers from Linux device drivers, 93 were concurrency related – mainly data races or deadlocks.

Mutilin et al. [11] found that data races are the most common single type of bug, remarkable 5 times more common than deadlocks in Linux device drivers.

It is worth noting that based on [3, 13], the average lifetime of a bug is 18 months, making it quite likely that the bug makes it way to a stable version.

In addition to being common, the bugs in device drivers can cause crashes fatal to the whole system. In Windows XP, 85% of the reported failures were caused by issues with device drivers [19].

The data races are extremely unpleasant in safety critical applications, where their presence could endanger lives. For that reason, the verification requirements of device drivers in safety critical domains (for example aviation or automobile industry) are very rigorous, making the testing process very costly and time-consuming.

### 3. DETECTING DATA RACES IN DEVICE DRIVERS

Not only are the data races in device drivers impactful, but also their non-deterministic nature means that common debugging practices do not work. It can be very hard to reproduce the issue with any kind of reliability and even when one manages to reproduce the issue, the information gained might not localize the issue in the codebase. Although common debugging practices are not time efficient when dealing with data races, they are still the most common way to look for data races, even when taking into account that a lot of tools that programmer working in the user-space are accustomed to are not available for debugging drivers running in Linux kernel-space, both because of the limitations of the domain and design choices [21].

This does not mean that tools that help programmers debug concurrency issues such as data races are inherently infeasible.

For example, `lockdep`, a runtime locking correctness validator added to the Linux kernel by Ingo Molnar [4], [20, Documentation/locking/lockdep-design.txt] in 2006 validates that the order in which locks are acquired is consistent. That is, if mutex $L_2$ was acquired while holding mutex $L_1$ at any point in the program, there cannot be a point where $L_1$ is acquired while holding $L_2$. When the order of acquiring locks is consistent throughout the program, then there cannot be a deadlock involving these two mutexes – a situation where thread $T_1$ is waiting to acquire $L_1$ while holding $L_2$ while thread $T_2$ is waiting to acquire $L_2$ while holding $L_1$.

With addition of `lockdep`, most of the bugs that could cause deadlocks have been eliminated from the kernel [5].

As debugging data races is hard and there are success stories in similar domains of tools that have proven to be invaluable, there has been quite a lot of research focused on tools that would help one detect data races. Vast majority of the results of this research make use of two key ideas.

### 4. KEY IDEAS FOR DATA RACE DETECTION

As discussed previously, it is possible that the same function of a driver is executed concurrently. Let thread template be the executable code reachable from one of the registered entry points of the driver. There can be any number of threads executing a single thread template at any point, corresponding to multiple calls to the same function of a device driver.

#### 4.1 Lockset analysis

Lockset analysis is a high-level idea about how to verify the absence of data races in a program. It relies on the assumption that the access to a shared variable should be governed by a lock.

Let us have two threads, $T_1$ and $T_2$ executing thread templates $t_1$ and $t_2$. Lets note that $t_1$ and $t_2$ do not have to be distinct.

Let $S$ be a shared state between those two threads, containing variables that are accessible from both threads and $L$ be a set of all locks that can be held by either thread.

Now, let $O^T_v$ be a set of all read or write operations of variable $v \in S$ in thread $T$. Lets define operation locks $L: O^T_v \rightarrow 2^L$, that for a read or write operation of variable $v$ in $T$ returns a set of all the locks that thread $T$ holds when that
operation is performed.

If set

\[
\left( \bigcap_{o \in O_T_1} \text{locks}(a) \right) \cap \left( \bigcap_{o \in O_T_2} \text{locks}(a) \right)
\]

is not empty, then that means that there is \( l \in L \) such that for every read or write operation of variable \( v \) in threads \( T_1 \) and \( T_2 \) is protected by \( l \) and as such, there is no chance for a data race. If such a lock exists for every variable \( v \in S \), we can soundly say that there is no possibility for a data race between threads \( T_1 \) and \( T_2 \) executing thread template \( t_1 \) and \( t_2 \). Note that this approach can be easily generalized for \( n \) threads and \( m \) thread templates.

Lets note that although this analysis is sound (we will not verify a program that has possibility of a data race), it is not precise, as there might be other ways than locking to make sure that no data race can take place.

4.2 Happens-Before

Happens-Before relation \( R \) is a partial-ordering of the statements, such that if \( a \) and \( b \) are statements then \( a, b \in R \) if and only if \( a \) is executed before \( b \). The concept was first introduced by Leslie Lamport [10].

For \( a \) to be executed before \( b \), either \( a \) has to be a statement before \( b \) in the same thread template, there must be a statement \( c \) such that \( a \) is executed before \( c \) and \( c \) is executed before \( b \) or there must be a synchronization event that is sufficient to say that \( a \) was executed before \( b \).

As an example of a synchronization event, lets consider two thread templates, in \( t_1 \) one acquires lock \( m \), reads a variable \( x \) and releases lock \( m \) and in \( t_2 \) one acquires \( m \), writes to \( x \) and releases \( m \). Here acquiring and releasing the lock works as a synchronization event – either reading variable \( x \) in \( t_1 \) happens before writing to \( x \) in \( t_2 \) or the other way around.

The key insight concerning data races is that if we can establish happens-before relationship between statements \( a \) and \( b \), then they cannot race against each other.

5. CATEGORIZATION OF THE TOOLS FOR DATA RACE DETECTION

There are quite a number of tools that make use of the two ideas described above. To be able to give a more systematic overview of them, we will first categorize them.

First categorization would be based on what the tools are meant to do. Some of the analysers aim to verify the absence of data-races. These analysers must be sound, they should never verify a driver where a data race is possible.

Others aim to ease testing of drivers and pointing out data-races to the programmers. In these cases, analysers should be mostly precise, i.e. they should not give false-positives.

While one would wish for a tool that is both precise and sound, the tools must function in the world of computational limitations and as of now, state of the art tools must make compromises.

Verifiers can in principle replace the laborious and expensive testing done by hardware manufacturers of safety-critical systems as of now, while tools focused on testing can only hope to improve the current process.

The analysers available can also be categorized as dynamic analysers and static analysers, depending on whether the analysis is performed during runtime or not. While static analysers take the source code of a program as input, dynamic analysers use the information gathered when executing the program.

Both approaches have their merits. While static analysers can be used independently of the program being analysed and as such, they do not compete for the resources with the test subject and can be computationally more complex, dynamic analysers can make use of the information available only during runtime.

Regarding verifiers, one should note that it is not possible to verify the lack of data races in a device driver with dynamic approach, as data races are famously hard to find because of the non-determinism they cause, meaning that one can not eliminate the possibility of a race based on a fixed amount of test runs.

6. STATE OF ART IN DATA RACES DETECTION IN DEVICE DRIVERS

We will now give an overview of notable tools that detect data races in device drivers.

6.1 KernelStrider

KernelStrider[18] is a dynamic analyser front-end for kernel space programs. It collects information about execution of device driver and forwards it to user-space. This makes it possible for one to use programs developed to detect data races in any C program to be used with device drivers. KernelStrider aims to make use of ThreadSanitizes[17], a dynamic data-race analyzer by Google for user-space, that makes use of both lockset analysis and happens-before. Another user-space dynamic analyser that KernelStrider output could be used with would be Helgrind ,[1, 12] that makes use of happen-before.

6.2 RaceHound

RaceHound[9] is a dynamic analyser for Linux device drivers. It borrows heavily from DataCollider [7], an analyser for Microsoft drivers. RaceHound places random software breakpoints to the driver under testing, then finds out which memory address was accessed and attaches a hardware breakpoint to that memory address and stops the thread for a while. When the hardware breakpoint is triggered by another thread, a data race has taken place.

The main benefit of this approach is the relatively low overhead of around 5% to the running time of the program under testing. This approach is precise, but not sound.

6.3 Locksmith
Locksmith[14] is a static analyser for data races in C programs, including device drivers. Locksmith was one of the earliest static analysers available for Linux device drivers and it validated the lockset based approach for static analyses.

For practically inclined, the authors of Locksmith share their experiences with static analysis for data races, describing the importance of identifying thread-local variables as a preprocessing step and for modelling fields of the structs fields sensitively and lazily[15].

6.4 Checkmate
Checkmate[8] is a generic static analyser for Java bytecode. It offers support for analyzing different runtime properties of multi-threaded program running on JVM, among them whether the program contains data-races.

Checkmate stands out by making extensive use of happens-before relation and not relying on lockset analysis for detecting data-races. The happen-before used by Checkmate over-approximates Java Memory Model.

6.5 Whoop
Whoop[6] is a static analysis tool for finding data races in Linux device drivers.

Whoop is a sound tool, making use of the lockset analysis. Whoop makes use of the observation that data-races happen between two specific threads and as such, it is enough to verify that there is no possibility of data-race between any pair of threads, instead of considering all the threads at the same time.

Whoop is meant to be used as part of a bigger tool-chain, where it functions as a preprocessing step to Corral, a bug-finder by Microsoft. During the preprocessing step, Whoop produces a sound model of the driver that can then be processed further by Corral.

The fitting together of a quite complex pipeline of tools to provide the analysis is a remarkable result of the project behind Whoop.

As of now, Whoop supports classical mutexes and spin-locks, but does not cover rest of the primitive locking options that are usable in Linux device drivers.

6.6 SDV and SLAM
SDV is a static analyzer by Microsoft for Windows device drivers.

As mentioned before, majority of the failures in Windows XP were caused by issues connected to device drivers. To solve this issue, Microsoft Research team started working on a way to verify the correctness of device drivers in early 2000s. The result of their work is Static Driver Verifier (SDV) that tries to verify a device driver during compile time.

A key part of SDV is SLAM [2], a tool that allows one to specify rules that must hold for device drivers and then statically check if they do for a driver during compile time. The main use case for SLAM in SDV is to confirm that drivers use the Windows driver API correctly.

SDV has been widely successful, with under 10% of reported issues being false reports [2, 74] and 97% of the runs resulting in either verification of the driver or finding a bug.

Although SDV does not focus on detecting data races, its success has shown that static analysis can be a viable approach for driver verification and it has had big influence in the field of static analysis of device drivers.

7. CONCLUSION
We gave a short overview of Linux device drivers and effects that data-races have in them. This paper also gave overview of static analysis tools relevant for detecting data-races in Linux device drivers and introduced the key ideas behind them.

8. REFERENCES


Deterministic parallel programming in Haskell

Ville Sokk
University of Tartu

Abstract. Parallel programming is error prone in comparison to traditional sequential programming but at the same time essential to use modern multi-core computers efficiently. This article will review three different Haskell programming models designed for guaranteed deterministic parallel programming: evaluation strategies, dataflow programming and data-parallel programming. These abstractions aim to make parallel programming more accessible, productive and safer.

1 Introduction

Parallel programs use multiple concurrent threads of execution to perform a task in less time than a traditional sequential program would take. Parallel and concurrent programming tends to induce more mistakes. The two main kind of mistakes are race conditions and deadlocks. A race condition happens when the result of a program depends on the timing of multiple threads accessing shared data. To prevent race conditions, mutual exclusion locks are the most common solution. They only allow one thread at a time to access the shared data. Deadlocks occur when threads need to acquire multiple locks and have to wait after each other. For example, two threads need to acquire locks A and B. If one of them acquires A and the other one acquires B then neither can proceed. Deterministic parallel programming models guarantee that the output of the program is the same every time it runs — no deadlocks can occur and the output can not depend on the scheduling of the threads. This article will look at three different models of guaranteed deterministic parallel programming in Haskell.

2 Haskell

Haskell is a pure functional strongly typed non-strict programming language. Haskell has powerful abstractions and separates pure computations from computations with effects. These features allow the programmer to extend the language with parallelism/concurrency constructs by creating a library as opposed to changing the compiler.

2.1 Monads

One feature of Haskell that can be used for extending the language is the concept of a monad which is borrowed from category theory. In Haskell a monad is a
type \( m \) (with a single type parameter) for which functions with the following signatures have been defined:

\[
\text{return} :: a \to m a \\
\text{bind} :: m a \to (a \to m b) \to m b
\]

A value \( m \ a \) represents a computation that returns a value of type \( a \). The function \( \text{return} \) takes a pure value and turns it into a computation returning that value. The function \( \text{bind} \) is used for combining two computations. The value of the first computation is available to the second.

Haskell has “do-notation” syntax which is similar to imperative programming languages and is translated into binds and returns. In the following code there are two examples of code written in the IO monad (which is used for input/output and otherwise imperative programming). The first one is written without do notation (the “\( >>= \)” operator is the inline bind function and “\( \backslash x \rightarrow \text{e} \)” is the anonymous function syntax). Both programs query the user’s name and print a greeting.

\[
\begin{align*}
\text{main1} :: & \text{IO ()} \\
\text{main2} &= \text{putStrLn "Please enter your name:" } \gg= \ \backslash x \rightarrow \\
& \quad \text{getLine } \gg= \ \backslash \text{name } \rightarrow \\
& \quad \text{putStrLn ("Hello " } \backslash + \ \text{name)}
\end{align*}
\]

\[
\begin{align*}
\text{main2} :: & \text{IO ()} \\
\text{main2} &= \text{do} \\
& \quad \text{putStrLn "Please enter your name:"} \\
& \quad \text{name } \leftarrow \text{getLine} \\
& \quad \text{putStrLn ("Hello " } \backslash + \ \text{name)}
\end{align*}
\]

The benefit of monads and do-notation is that by defining monads (defining return and bind for a type) the programmer can repurpose do-notation. In a sense, the programmer defines what it means to sequence two statements in an imperative programming language. The following example uses parser combinator of the Parsec library (an embedded domain specific language for writing parsers which uses monads) to define a parser for parsing “key=value” pairs separated by line breaks.

\[
\begin{align*}
\text{string} :: & \text{Parser String} \\
\text{string} &= \text{many alphaNum} \\
\text{entry} :: & \text{Parser (String, String)} \\
\text{entry} &= \text{do} \\
& \quad x \leftarrow \text{string} \\
& \quad \text{char ’=’} \\
& \quad y \leftarrow \text{string} \\
& \quad \text{return } (x, y)
\end{align*}
\]

\[
\begin{align*}
\text{dict} :: & \text{Parser [(String, String)]} \\
\text{dict} &= \text{sepBy entry (char ’\n’)}
\end{align*}
\]
2.2 Laziness

Haskell is a non-strict programming language which means evaluation of expression starts from the outermost expression, not from sub-expressions. In the most widely used compiler GHC (Glasgow Haskell Compiler), this is achieved through lazy evaluation. Lazy evaluation means that expressions are only evaluated when they are needed. For example, in the following program, \( x \) (an infinite list of natural numbers) is never evaluated because the program’s output does not depend on it.

```haskell
main :: IO ()
main = do
  let x = \[1..\] :: \[Integer\]
  putStrLn "Hello"
```

The following program prints the first ten elements of \( x \). When evaluating the `print` procedure, values of the list are computed one by one on demand and interleaved with `print`'s own computations (like printing the commas). Values after the first ten are not computed.

```haskell
main :: IO ()
main = do
  let x = \[1..\] :: \[Integer\]
  print (take 10 x)
```

2.3 Runtime system

The runtime system of GHC uses user space threads. Instead of using hardware interrupts and the operating system’s scheduler, a user space thread executes code until it reaches some point where it voluntarily passes control to the scheduler of the runtime system. Avoiding context switches between the operating system and user space allows fast switching of threads. Since user space threads run in the same kernel thread, they can not take advantage of multiple cores. To utilise multi-core computers, the GHC runtime system can use a small number of kernel threads (for example, one per core) which each execute a set of user space threads. Even though user space threads can not use hardware interrupts and control has to be relinquished by the thread itself, the programmer is not concerned with this because the GHC compiler implicitly adds yields of control.

This model has proven to be efficient while being easy to program. A significant number of threads can be managed efficiently so the programmer does not have to be concerned about creating too many threads or using thread pools. For example, a Haskell web server called mighty achieves performance close to or better than nginx which is considered to be one of the fastest web servers\[11, 3, 2\]. Mighty uses Haskell threads that are programmed sequentially like POSIX threads while nginx is more complicated and schedules requests by itself.
3 Evaluation strategies

3.1 Introduction

Evaluation strategies allow the programmer to declare which expressions in their code should be evaluated in parallel and define methods of parallel evaluation for different data structures [8]. Strategies are implemented in a library named parallel.

Let’s first look at a simple monad for evaluating expressions in parallel called Eval. It provides two functions `rpar :: a -> Eval a` and `rseq :: a -> Eval a`. The expression `rpar e` adds a pointer to the unevaluated expression `e` in a data structure called the spark pool. Kernel level threads of the runtime system can pick sparks from the spark pool and evaluate them in a user space thread. The expression `rseq e` evaluates expression `e` in the thread calling `rseq e`. Since Haskell is lazy, `rseq` is necessary to define the order of evaluation. Note that `rpar` does not create a thread. The argument of `rpar` may or may not be picked from the spark pool and evaluated in a parallel thread. So strategies create possibilities for parallel evaluation but they cannot be used to explicitly distribute work between a number of threads. The following example is a parallel implementation of the fibonacci function using the Eval monad:

```haskell
fib :: Int -> Int
fib n = runEval (do x <- rpar (fib (n - 1))
                     y <- rseq (fib (n - 2))
                     return (x + y + 1))
```

`runEval` has type `Eval a -> a` and is used to run a parallel computation. The line `x <- rpar (fib (n - 1))` creates a spark with the expression `fib (n - 1)`. The variable `x` will eventually be bound to the value of the call `fib (n - 1)`. The current thread then continues with `y <- rseq (fib (n - 2))`. The call to `rseq` forces the thread to evaluate `fib (n - 2)` before continuing to the next line. Now, in `x + y + 1`, `y` is evaluated, but `x` may not be evaluated. If, during the time the current thread was evaluating `y`, another thread picked up the spark of `x` and finished evaluating it before `y` was evaluated, its value is known and `x + y + 1` can be computed. Otherwise, the current thread has to evaluate `x` by itself.

The Eval monad is used to implement strategies. A strategy defines how to evaluate a data structure in parallel. The type of a strategy is type `Strategy a = a -> Eval a`. For example, we can define a strategy for pairs:

```haskell
parPair :: Strategy (a, b)
parPair (a, b) = do a' <- rpar a
                     b' <- rseq b
                     return (a', b')
```

It sparks the first possibly unevaluated element so that it could be evaluated in another thread and evaluates the second element in the current thread. The pair is then reconstructed using the evaluated elements.
Strategies can be combined to create strategies for evaluating more complicated structures. For example, the \texttt{parList} function takes a strategy for type \texttt{a} and uses it to create a strategy for evaluating elements of a list \([a]\) in parallel:

\begin{verbatim}
  evalList :: Strategy a -> Strategy [a]
  evalList s [] = return []
  evalList s (x:xs) = 
    do x' <- s x
       xs' <- evalList s xs
    return (x':xs')

  parList :: Strategy a -> Strategy [a]
  parList s = evalList (rpar 'dot' s)
\end{verbatim}

\texttt{evalList} just applies a strategy to each element of a list. The extra \texttt{parList} call composes the given strategy \texttt{s} with \texttt{rpar} so that a spark would be created for each list element (\texttt{dot} is the function for composing strategies like the function composition operator \texttt{.}).

A convenience function \texttt{withStrategy} is provided that uses a supplied strategy to evaluate an expression in parallel:

\begin{verbatim}
  withStrategy :: Strategy a -> a -> a
  withStrategy s x = runEval (s x)
\end{verbatim}

There are some built-in strategies that work with all types. The strategy \texttt{r0} performs no evaluation and is equivalent to a regular lazy single threaded Haskell program. The strategy \texttt{rseq}, which we have already covered, evaluates its argument to weak head normal form. This means that the outermost constructor is evaluated. For example, if the type of an expression is \((\texttt{Int}, \texttt{Int})\), and the expression is unevaluated, we have a pointer to the expression. If it is evaluated to weak head normal form, we have a pointer to \((\texttt{e1}, \texttt{e2})\) where \texttt{e1} and \texttt{e2} are pointers to the expressions with type \texttt{Int}. If it is fully evaluated, we could have a pointer to something like \((1, 2)\). The third strategy is \texttt{rdeepseq} which fully evaluates its argument.

Now, if we have a number of strategies for parallel evaluation of different data structures, we can use them and \texttt{withStrategy} to parallelise our sequential programs. Let’s define a parallel \texttt{map} function.

\begin{verbatim}
  parMap :: Strategy b -> (a -> b) -> [a] -> [b]
  parMap strat f list = withStrategy (parList strat) (map f list)
\end{verbatim}

\texttt{parMap} takes a strategy for evaluating list elements, a function and a list. It maps the function to each element and then uses the previously defined \texttt{parList} to evaluate the transformed list in parallel. Let’s define a function for multiplying matrices using the naïve algorithm.

\begin{verbatim}
  dotProduct :: Num a => [a] -> [a] -> a
  dotProduct a b = foldl' (+) 0 (map (\(x, y\) -> x * y) (zip a b))

  columns :: [[a]] -> [[a]]
  columns rows = transpose rows
\end{verbatim}
matrixMultiplication :: [[Int]] -> [[Int]] -> [[Int]]
matrixMultiplication a b =
  map (\aRow ->
      map (\bCol -> dotProduct aRow bCol)
        (columns b))
  a

  We can now efficiently parallelise it by just changing the outer map to
  parMap rdeepseq.

parMatrixMultiplication :: [[Int]] -> [[Int]] -> [[Int]]
parMatrixMultiplication a b =
  parMap rdeepseq (\aRow ->
      map (\bCol -> dotProduct aRow bCol)
        (columns b))
  a

  It is obvious that parMap creating a spark for each element is very inefficient
  if there are a lot of elements. To use coarser granularity, the library provides
  a function called parMapCluster which takes a cluster size, splits the data
  structure into smaller clusters and evaluates the clusters in parallel.

  In some parallel algorithms, a producer/consumer model is used where a
  consumer process consumes a data structure produced by a producer process.
  The data can be produced on demand and stored in an intermediate buffer. A
  function called parBuffer is provided for this model.

parBuffer :: Int -> Strategy a -> Strategy [a]
parBuffer n s list creates n threads evaluating the first n elements
  of list. When the consumer demands the first element of the list, parallel
  evaluation of the nth element is sparked. So as the consumer consumes the
  (possibly infinite) list, up to n threads are precomputing its upcoming values.

  Purity means that a function call f(c) where c is a constant, always results
  in the same value. We can understand intuitively that it does not matter in
  which order the values of f’s arguments are calculated as long as we know the
  values before calculating the value of the function call. Since strategies are pure,
  computations using strategies are deterministic. We can not write to shared
  memory, cause a deadlock or observe the order of computations in any way.

3.2 Discussion

Using parallel strategies we can write a sequential version of the program first
and then parallelise it without obfuscating the logic of the program.

  The user of the library rarely has to write a strategy. In Haskell, if a new
  data structure is defined, an implementation of the traverse function in the
  Data.Traversable module of the standard library is usually provided. This
  function defines how to traverse the data structure and perform some monadic
  computation on each value and then collect the results into the same type of
data structure with the same structure. For instance, traverse could be used to print out the values of a list and increment numbers in a tree. This means traverse is polymorphic in the type of data structure (in this case list or tree) and in the type of side-effects (in this case I/O or no effect). There is a generic strategy called parTraverse that uses traverse to define how to evaluate a data structure in parallel. So strategies are usually provided “for free” for each data structure.

It is clear that this method has significant limitations as well. Different threads sparked by rpar have no means of communicating with each other. Parallel algorithms exist which are not expressible using strategies. Strategies rely on laziness. For example, the producer/consumer model using parBuffer relies on an intermediate lazy data structure (e.g. a lazy list). To write efficient programs using strategies, the user must have a good understanding of the lazy evaluation model used by GHC. Finding and debugging performance issues caused by laziness can be difficult. For example, the sum of a list of integers can be computed using foldl (+) 0 xs. This uses more memory than necessary because it creates a thunk with the expression 0+x_1+x_2+...+x_n (where x_i is the i-th element of xs) instead of adding the elements. The problem can be easily fixed by writing a version of foldl that forces the computation of the accumulator argument every time it is called. But it is difficult to understand when laziness causes excess memory usage or when it is necessary for parallel evaluation.

4 Accelerate

4.1 Introduction

Data parallelism is a model of parallel programming where the data is distributed to tasks and each task performs the same operations. Accelerate is a Haskell library which provides a multi-dimensional array type and a pure embedded domain specific language (EDSL) for programming computations on arrays [1]. The EDSL is compiled into a parallel program on a target platform. The library supports different backend compilers for CPUs and GPUs (graphics processing units). A programmer writing an Accelerate program will use high level combinators similar to the ones used with lists, for example map, fold, filter. The combinators are relatively high level and can be implemented using parallel algorithms. For example, in a map operation, sub-arrays can be distributed to threads which can perform the map independently. A fold can be parallelised when the folding operation is associative. Sub-arrays can be folded in parallel and the resulting sub-array can be folded again. Accelerate’s support of multiple compiler backends means that the high-level combinators can have different parallel implementations on different platforms to achieve the best performance on each platform.

Accelerate arrays are multidimensional. The type of arrays is Array sh e where sh is the shape type and e the element type. For example, matrices
indexed by integers have the shape $Z :: \text{Int} :: \text{Int}$, vectors indexed by unsigned integers have the shape $Z :: \text{Word}$ and scalars (0-dimensional arrays) have the shape $Z$.

The following example computes the dot product using Accelerate.

```haskell

dotp :: Acc (Vector Float) ->
  Acc (Vector Float) ->
  Acc (Scalar Float)
dotp xs ys =
  fold (+) 0 (zipWith (*) xs ys)
```

Vector $a$ is a shorthand for Array ($Z :: \text{Int}$) $a$ and Scalar $a$ is shorthand for Array $Z a$. The Acc (Vector Float) type can be read as “an Accelerate expression that evaluates to a vector of floats”. Practically, Acc is the abstract syntax tree type of the language. We can see that the code very clearly expresses the definition of the dot product: it is the sum of the products of the respective elements of the input vectors. Nothing is expressed about parallelism. This code can be compiled into a parallel GPU or CPU program by inserting the (+) and (*) expressions into the parallel fold and zipWith algorithms.

Let’s look at one way of doing naïve matrix multiplication.

```haskell

matrixMultiplication :: forall a. (IsNum a, Elt a) => Acc (Array DIM2 a) -> Acc (Array DIM2 a) -> Acc (Array DIM2 a)
matrixMultiplication a b = result
  where
    Z :: m :: n = unlift (shape a) :: (Z :: (Exp Int) :: (Exp Int))
    Z :: _ :: p = unlift (shape b) :: (Z :: (Exp Int) :: (Exp Int))

    tmpA = genTmp a (\m p n -> Z :: m :: n)
    tmpB = genTmp b (\m p n -> Z :: n :: p)

    genTmp mat mkIdx =
      generate (lift (Z :: m :: p :: n))
      (\ix ->
        let Z :: m :: p :: n = unlift ix :: (Z :: (Exp Int) :: (Exp Int))
        in
          mat ! lift (mkIdx m p n))

    tmp = zipWith (*) tmpA tmpB

    result = fold (+) 0 tmp

```

Array indices are constructed using the same constructors as the types. So $Z :: 1 :: 1$ is the index $(1,1)$ of a matrix.

The binding $Z :: m :: n = unlift (shape a)$ reads the dimensions of matrix $a$ into the variables $m$ (rows) and $n$ (columns). The type of shape
a is Exp (Z :. Int :. Int). The type Exp is another type representing Accelerate computations (the difference between Acc and Exp will be explained later). So in the type of shape a, Z is inside Exp and we can not pattern match on it. This is understandable because we are not dealing with Haskell values. The shape of the matrix depends on runtime data and the code might be running on the GPU and not in Haskell at all. The function unlift turns it into a value with type Z :. (Exp Int) :. (Exp Int) which means that we again do not know the actual values but we now have computations representing them. Note that we give the type explicitly using the var :: type syntax because type inference fails here. Now we know that the first matrix has shape (m, n) and the second has shape (n, p) so the result must have shape (m, p).

We generate two temporary matrices with shape (m, p, n). They can be imagined as matrices whose elements are n-element vectors. The vector at (i, j) of tmpA is the vector from a that is used in the dot product of element (i, j). The expression mat ! idx is used for indexing and idx can be multi-dimensional. The expression generate shape f creates an array with shape shape and passes the index of each element to f which outputs the value at that index.

After we have calculated the intermediate matrices, we can multiply the dot product vectors element-wise using zipWith (*) tmpA tmpB. Note that zipWith works on the innermost dimension. Finally, fold (+) 0 tmp sums each dot product vector (fold also works on the innermost dimension).

Clearly, this is nothing like the straightforward Haskell code we used when discussing parallel strategies. We can not use nested map functions like we did earlier. To understand why, let’s look at the type of map.

map :: (Shape ix, Elt a, Elt b) =>
    (Exp a -> Exp b) ->
    Acc (Array ix a) ->
    Acc (Array ix b)

Let’s ignore the constraints before =>. The first argument is Exp a -> Exp b which is like a function a -> b in Haskell. The second argument is an Acc expression whose value is an array with elements of type a and the result is an Acc expression whose value is an array with elements of type b. Both Acc and Exp are used to represent computations in the EDSL. The difference is that the result of a call to a combinator like map is an Acc expression but the function arguments of combinators are always Exp expressions. This makes it impossible to pass a function that uses a combinator to another combinator. If this was allowed, it would lead to nested data parallelism (map creates threads and each thread creates more threads by calling the supplied function). Efficiently compiling nested data parallel programs to GPUs is an open research problem so it is currently statically prohibited in Accelerate. But this is rather limiting. The function passed to a combinator can not use other combinators, it also can not use recursion and there are only very limited built-in iteration constructs. We might be able to program a simpler version of the matrix multiplication but not using nested map calls as we did before.
A naïve compilation of this kind of programs would result in a large number of intermediate arrays. The matrix multiplication example created three unnecessary temporary matrices. Accelerate uses an optimisation called fusion which can eliminate intermediate arrays by combining successive traversals [9].

Writing programs in Accelerate is not completely like traditional functional programming. For example, an efficient primitive in Accelerate is \texttt{permute}. The use of \texttt{permute} is demonstrated in the following histogram calculation program (elements are expected to be in the range \([0, 100]\)) [9].

\begin{verbatim}
histogram :: Acc (Vector Float) -> Acc (Vector Int)
histogram vec =
  let bins = 10
      zeros = fill (constant (Z :. bins)) 0
      ones = fill (shape vec)
      mapping ix = index1 (A.floor ((vec ! ix) / P.fromIntegral
                                  bins))
in
      permute (+) zeros mapping ones
\end{verbatim}

\texttt{mapping} is a function that maps an index of the input array to an index of the output array. In this case it calculates the index in the output vector based on the value in \texttt{vec}. Since some values of the output vector may not be covered by the index mapping, default values must be provided (\texttt{zeros}). Multiple input indices may be mapped to the same output index (which is expected in this example) so a function must be provided that can combine multiple elements that are mapped to the same index in the output. In this case the input vector to \texttt{permute} is a vector of ones and the combining function adds ones that are mapped to the same output index so it counts the number of values in a histogram bin. This primitive is very powerful and can be used to implement other primitives like \texttt{filter} but it is not a familiar combinator that is used with lists.

4.2 Discussion

One of the benefits of Accelerate is support of different platforms and runtime compilation. In some cases, programs must work in heterogenous environments. For example, software distributed to end users with home computers. The program should work relatively efficiently for everyone, including users with GPUs from different manufacturers and users without a GPU. One backend compiler of Accelerate also supports the compilation of a single program to a heterogenous set of devices [10]. In scientific computing clusters, computer nodes have multi-core CPUs and GPUs. The backend compiler can transform a program into multiple programs running on both the CPUs and GPUs of a computer and split the input data among the devices.

Since the code is compiled during runtime, it can be optimised depending on some dynamic values. For example, user chosen parameters of an algorithm or specifications of the device.
Since the high level code remains the same on different platforms, the developer can test their code even when they have no access to the target platform. Accelerate is a high level language. Using a low level language and optimising your program for the target platform can result in better performance. In [9], performance of Accelerate programs compiled with the CUDA backend is compared to hand optimised CUDA programs. Accelerate programs can get close but maximum performance is usually achieved by hand optimised CUDA programs.

If the programmer wants their program to be efficient, it often does not suffice to write the most straightforward and clear version of their program. Optimisation is still required. To extract maximum performance, the programmer must understand both how the target platform works and how Accelerate works. For example, to write a program that iteratively improves an estimate, multiple instances of the estimation expression are composed using the pipeline operator \( \rightarrow \). If it is used repeatedly, the resulting code can be big, it can take some time to compile (remember that the program is compiled during run time) and the whole program may become too big for the instruction cache of the target platform which will reduce performance.

We saw that the language is very restricted which allows it to be efficiently compiled to different platforms. The result is that code written in Accelerate is often very different from a regular functional program and the programmer has to spend effort to formulate their problem within the restrictions of the Accelerate framework.

5 LVars

5.1 Introduction

The LVars model allows communication through shared monotonic data structures to which information can only be added, never removed, and for which the order of updates is not observable [6]. LVars are implemented in the LVish library.

Similar to strategies, LVars have a monad called \texttt{Par} for parallel programming. The monad allows the programmer to create, write and read LVars and create threads that run \texttt{Par} computations. To create a thread, a monadic computation is provided to \texttt{fork} (simplified type):

\[
\texttt{fork} :: \texttt{Par} () \to \texttt{Par}()
\]

A \texttt{Par} computation can be run using \texttt{runPar}:

\[
\texttt{runPar} :: \texttt{Par} a \to a
\]

The simplest form of an LVar is IVar which existed (in the monad-par library) before the LVars model was conceived. An IVar is either empty or full and can only contain a single value. A program using IVars describes a dataflow graph where information flows between nodes (threads) and where edges are the IVars used for communication. For example, in the following program, assume that \( e_1 \) and \( e_2 \) are some expressions with a numeric value:
runPar
  (do x1 <- new
      x2 <- new
      fork (put x1 e1)
      fork (put x2 e2)
      r1 <- get x1
      r2 <- get x2
      return (r1 + r2))

The corresponding dataflow graph is:

```
   e1 + e2
     /    \
    /     \
   e1   e2
```

We can use IVars to program a parallel matrix multiplication again:

```haskell
parMatrixMultiplication :: [[Int]] -> [[Int]] -> [[Int]]
parMatrixMultiplication a b =
  runPar
  (do vars <- forM a $ \aRow -> do
      var <- new
      fork (computeRow aRow var)
      return var
    )
  where
    computeRow row var = do
      let res = map (\bCol -> dotProduct row bCol) (columns b)
      put var res
```

For each row in the matrix a, an IVar is created with new and a thread is created with fork which computes the row and writes the result into the variable. The variables are collected into the vars list. For each variable, mapM get vars waits until its value has been written to the variable and retrieves it. Each IVar is written only once and can thus contain only one value. The value is not dependent on when it is read – if it has been written it stays constant and if it has not been written then get blocks until it is. Since the computations are otherwise pure, no information from the outside world can be written into the IVars which could cause non-determinism. Unfortunately, the write-once policy is not guaranteed statically but a runtime error occurs if multiple puts to the same IVar are executed with different values. If the write-once policy is followed, the result of the computation does not depend on the scheduling of the threads.

LVars are a generalisation of IVars. There’s a set of states that an LVar can be in. These states must form a lattice – a partially ordered set where each two elements have a least upper bound and a greatest lower bound. LVars can be read and written but not like a regular mutable variable. The write operation takes the least upper bound of the current state and the state that the LVar
is being transitioned to. This ensures that the information in the variable only accumulates, it cannot be reduced (writes can only move up the lattice). Only threshold reads can be performed on LVars—a thread can block until an LVar reaches a state in a threshold set. The states of a threshold set must be pairwise incompatible—their least upper bound must be top (the error state that is “greater than” all states). The reading thread can only observe that the LVar state has reached a threshold state (or gone further) but it cannot observe the order of writes that lead to this state.

An example of an LVar is a variable containing a natural number. Natural numbers have a total ordering and thus form a lattice. The variable would start out empty. The write operation would take the maximum of the current value and the value being written. A threshold read would be a thread specifying a number (the singleton set containing that number is the set of threshold states) and blocking until the value in the variable is greater than or equal to the number.

The fact that writes only increase information, reads are threshold reads and threads have no other side effects except communication through LVars and creating new threads, guarantees that the result is not determined by the scheduling of the threads.

Although LVars are more limited than regular mutable variables, a number of useful concurrent data structures can be programmed in this model. Virtually any data structure to which information is added gradually can be represented as a lattice, including pairs, arrays, trees, maps, and infinite streams [5]. For example, the LVish library provides maps and sets that support insertions and lookup (but not modification or removal). To insert a variable to a set, the least upper bound would be the union between the current set and a singleton set containing the added element. A threshold read would take a value and wait until the set contains that value. Maps could likewise be merged when writing and a threshold read would take a key, wait until a value with that key appears and return the value. Only the implementer of a new LVar has to know what a lattice is and must follow the restrictions imposed on write and read. Most programmers should be able to use predefined concurrent data structures implemented in the LVars model.

The basic LVar model has been extended to make it possible to solve more problems. The requirement that states in a threshold set are pairwise incompatible is too strict. For example, to have a parallel conjunction operation (as in implicitly parallelised logical programming), we can represent the state as a pair of two inputs. Each input is either bottom (no input yet), true or false. The threshold set consists of all the states where both inputs are available—\{ (true, false), (false, true), (true, true), (false, false) \}. But this would not allow short circuiting (early return when one input is false). A more relaxed condition would allow multiple sets of pairwise incompatible states. Using this condition the threshold set becomes \{ (false, bottom), (bottom, false), (false, true), (true, false), (false, false), (true, true) \}. The threshold read now returns the set that “activated”. In the
first case, the result of the conjunction is false (even if one input has not been computed yet) and in the second case the result is true.

The `put` function is also more restrictive than necessary. For example, it is not possible to increment a number in parallel. It is possible to write an integer variable in parallel (as described earlier) but since the exact state of an LVar can not be read, it is not possible to increment the current value in the variable. A relaxed model allows a set of update operations that meet two conditions. When an update operation is applied to a state, the resulting state is greater or equal to the input state according to the ordering. When two update operations are applied in sequence, the result is not dependent on the order of the operations. Incrementing is safe as it moves the state upwards in the ordering of integers and the order of increments does not matter.

The `get` operations are blocking – they wait until a state in the threshold set is reached. Event handlers extend LVars to asynchronous programming. Given a set of states called an event set, an event occurs when an LVar reaches a state in the event set or a state that is greater than a state in the set. An event handler is a callback function that is called in a worker thread of the LVish library when an event occurs. The handler is called once for each state in the set that has been activated (for each state less than or equal to the current state). For example, if we have an integer LVar `lv` and we add a handler that is activated on the event set `{1, 2, 3}`, then `put lv 2` causes the handler to be called with argument 1 and 2, because both event states are less than or equal to the state of the LVar. Event handlers are also `Par` computations so they can cause other handlers to be fired by using `put` operations.

The function `freeze` allows an LVar to be frozen (`puts after freeze will raise an exception`) and its final state to be read directly. It is not deterministic though. If a `put` occurs after `freeze`, an exception is raised. If the `put` occurs before `freeze`, the computation finishes successfully. A computation using `freeze` is guaranteed to be quasi-deterministic. A quasi-deterministic computation always produces the same result if it finishes, or throws an exception if it does not finish. In the LVish library, quasi-deterministic and deterministic computations are separated on the type level. A parallel computation returning an integer has type `Par Det Integer` if it is deterministic and `Par QuasiDet Integer` if it is not (the full type in the LVish library has a third parameter which is not relevant in this discussion). Calls to `freeze` do not typecheck in `Par Det a` computations. This allows the programmer to still have the determinism guarantee if they do not use `freeze`.

Event handlers can be added to an event handler pool. A function called `quiesce` takes a handler pool and blocks until all event handlers have finished firing. While `freeze` itself is only quasi-deterministic, the combination of `freeze` and `quiesce` gives a useful function called `freezeAfter`. The function call `freezeAfter lv Q f` takes an LVar `lv`, an event set `Q` and a handler `f`. It creates a handler pool, adds the handler, uses `quiesce` to wait until the handler has finished firing and then freezes the LVar. Freezing after `quiesce` is deterministic if no further `puts` occur. The pattern of freezing after `quiesce`
is used for fixpoint computations (iteratively building a data structure until it stops changing) which can solve problems that the parallel strategies and Accelerate models did not solve. For example, the following is an example of parallel graph traversal [5].

\begin{verbatim}
traverse g startNode = do
  seen <- newEmptySet
  h <- newHandler seen
  (\node -> do
    mapM (\v -> insert v seen) (neighbours g node)
    return ())
  insert startNode seen
  quiesce h
  freeze seen
\end{verbatim}

g is the graph. First, a monotonic set is created using newEmptySet that will collect visited nodes. A handler is created for this set. The set API is designed so that this handler is called every time an element is added to the set. The handler then adds all neighbours of the node to the seen set which causes the handler to be fired again. The expression insert startNode seen inserts the first node which starts the computation. The expression quiesce h blocks until the handler stops firing and freeze seen returns the final set. Using freeze and quiesce is not deterministic, but this combination is and the author of a monotonic data structure should define a function for this pattern. Fix-point computations such as this can be used for a number of useful graph problems such as finding minimum spanning trees.

Kuper also describes how the model can be extended with thread local mutable state [5]. When threads are created, a mutable data structure can be split so that each created thread sees only a subset of the data structure. These subsets are disjoint and the typechecker can assure that different threads can not access each others’ state. For example, this can be used to split an unboxed vector among threads which then use local mutable state to update their slice. This is almost like imperative programming while preserving determinism.

Note that the lattices describe the semantics of LVars. An implementer of a monotonic data structure does not have to program with threshold and query sets explicitly. In fact, an event set can be infinite, as in the case of the set callback whose event set consists of all singleton elements. The implementer of an LVar must make sure that puts and gets are monotonic. Only a safe API which is not concerned with lattices, is provided to the programmer. In the case of sets, the simplest API would allow the programmer to create a set, insert elements and wait until a specific element appears in the set.

5.2 Discussion

Concurrent data structures with a guaranteed deterministic programming interface can be implemented using the LVars model and the thread scheduling infrastructure of the LVish library. LVar event handlers can be used to implement
pipelined computations. An event handler would push data to the next stage using a `put` operation. By supporting inter-thread communication, thread-local state and event handlers, LVars can be used to implement a wide variety of concurrent data structures. These data structures can be used to implement different complicated parallel algorithms. Most programmers do not need to implement their own data structures. The data structures have a familiar interface to their non-concurrent counterparts. They are only missing destructive updates so the transition from a single threaded program to a multithreaded program will not be too difficult.

One possible issue with LVars is that calling event handlers on each state in an event set will cause a lot of unnecessary work. For example, in the case of sets, adding an element to a set would cause an event handler to be called for each element in the set.

Optimising LVar programs could be difficult. For example, a programmer who does not know about the semantics of LVars and who uses an event handler may not be aware of how event handlers are called multiple times for each element. Event handlers cause further events if `puts` are used in the handlers. A cascading effect occurs and even though the programmer can understand that their program is semantically correct, they might be confused about the runtime behaviour.

6 Comparison

The Accelerate model is the most restricted. Only non-recursive functions on scalars can be mapped over arrays (although some iteration constructs are provided). For example, this makes it difficult to write a sequential sorting algorithm and apply it to columns/rows of a matrix in parallel which is used in parallel nearest neighbour search on GPUs [4]. Accelerate provides no means of communicating between threads. It is difficult to write parallel programs where threads build a shared data structure (as in the graph traversal problem).

Parallel strategies are less restricted because the programmer can use arbitrary pure functions. Parallel strategies allow the programmer to easily parallelise existing Haskell code by declaring which lazy structures should be evaluated in parallel. But parallel strategies have no user visible threads and can not provide any way to communicate between threads. Strategies rely on lazy evaluation which can be difficult to understand. In many cases, lazy data structures are inefficient and we may want to use a strict data structure such as unboxed vectors. Values of unboxed vectors are stored in memory sequentially so an unboxed vector can not be lazy. Thus programs relying on unboxed vectors can not be parallelised using strategies. The fact that strategies only create opportunities for parallelism (by creating sparks which can be evaluated in different threads) but there is no guarantee that sparks are evaluated in parallel, can cause further confusion. Let’s imagine parallelising the following graph traversal example [7]:

explore :: Set State -> [State] -> Set State
explore seen [] = seen
explore seen (todo:todos)
  | member todo seen = explore seen todos
  | otherwise = explore (insert todo seen)
    (toList (next todo) ++ todos)

We would like to replace (next todo) with something like withStrategy
(parBuffer 8 rseq) (next todo) which would expand the list of unvisited
nodes in parallel. This will not work because the recursive call to explore
will immediately check if the first element of the unvisited nodes list is in the
seen set. The spark that should be computing the to-be-visited node is then
removed from the spark pool because the main thread will compute it instead.
Sometimes, adding parallelism using strategies may result in no speedup at all.
It can be difficult to understand why it does not work as intended. Creating
threads explicitly and using variables for communication as in the LVars model,
is easier to reason about.

The LVars model is the least restrictive by providing communication between
threads through LVars, asynchronous event based programming and thread local
state. The LVars model can be used to implement a lot of different concurrent
data structures with a familiar but deterministic interface.

LVish and parallel strategies only work on CPUs. Accelerate code is com-
piled during runtime and can use hardware like SIMD instructions or coproces-
sors (GPUs, Intel Xeon Phi) if the necessary backend compilers are developed.
Accelerate also performs optimizations on the users code like fusing successive
stages of computation so it could theoretically provide better performance.

An important issue with all of the models is the difficulty of reasoning about
runtime behaviour. While determinism is guaranteed, efficiency is not. The goal
of parallel programming is to improve efficiency. If a programmer is not able
to understand why their program is performing poorly, they will not be able to
extract maximum performance.

When maximum performance is required, it is still best to use unsafe methods
like locks. In Haskell, the MVar data structure is a combination of a shared vari-
able and a lock. Threads that run an IO monad computation can be created with
forkIO. MVar provides no determinism guarantee but the runtime behaviour
of a program using forkIO and MVars is easier to understand. If a programmer
wishes to parallelise an existing Haskell program with relatively little effort and
does not want to spend time debugging and optimising for maximum performance,
LVish and parallel strategies are a great opportunity. If an application
domain uses different devices (CPUs and GPUs), the Accelerate model can be
used to implement a program once and compile it for different platforms.

References

C# and C++ interoperability

Andres Traks
Tartu University
Institute of Computer Science

Abstract. C# is a modern programming language aimed at code robustness and development productivity, but it can't compete with C++ in performance. The best of both worlds can be had by interoperating between the two languages.

The challenge lies in the fact that C# follows a different paradigm than C++ in many ways. For example, C# cleans up memory using automatic garbage collection while C++ requires memory to be freed explicitly. Low-level memory access is natural in C++, but is strictly controlled in C#. Not to mention differences in naming conventions and semantics.

This paper describes several approaches to bridging the gap between C# and C++ and how to overcome performance and memory management issues.

By combining the power of C# and C++, developers can build their application in a safe and productive manner without sacrificing speed in performance-critical parts of the code.
Table of Contents

1 Introduction ............................................................... 3
  1.1 Motivation ........................................................... 3
  1.2 Interoperation goals ............................................... 3
  1.3 Types of interoperation ........................................... 3
  1.4 Binary interface of DLL files ..................................... 6
2 Mapping of language features between C++ and C# ............... 6
  2.1 Basic types .......................................................... 7
  2.2 Classes and access methods ....................................... 7
  2.3 Fields ................................................................. 9
  2.4 Resource cleanup .................................................. 10
3 Mapping of coding conventions between C++ and C# .............. 12
  3.1 Capitalization ........................................................ 12
  3.2 Prefixes ............................................................... 12
  3.3 Inline documentation .............................................. 13
4 Automated wrapper library creation .................................. 13
  4.1 AST parsing .......................................................... 13
5 Future work ............................................................. 14
6 Glossary ................................................................. 14
1 Introduction

1.1 Motivation

C# is not designed to compete with C++ in terms of performance despite continuous improvements in the .NET framework and the C# compiler. C++ can more easily take advantage of CPU features such as Advanced Vector Extensions (AVX) and Streaming SIMD Extensions (SSE). C++ also supports compile-time optimizations such as aggressive code inlining and inline assembly.

When working in C#, it is therefore often more practical to interoperate with existing C++ libraries rather than porting them to C#, especially in areas requiring high processing throughput such as computer games and high-performance computing.

1.2 Interoperation goals

Our aim is to wrap an existing C++ library with a .NET wrapper library, which will be consumed by a .NET program.

The wrapper library defines intermediate functions that use .NET semantics and conventions. These functions will then provide access to the functions in the C++ library.

An example of differing conventions is that the C++ language does not specify the capitalization of function names, so there could be a function name starting with a lowercase letter (“add”). In C#, capitalization is enforced by Common Language Runtime (CLR) conventions, which state that functions should use Pascal case (“Add”).[1]

By understanding these differences, we can formalize the mapping between C++ and C# and move towards automatically generating wrapper code for C++ libraries.

1.3 Types of interoperation

There are several ways to call C++ code from C#. Each method has trade-offs. This article describes the C++/CLI and Platform Invoke methods in detail.

C++/CLI  C++ on Common Language Infrastructure (C++/CLI) is Microsoft’s programming language designed to extend C++ with CLI types.[3] C++/CLI can mix both unmanaged C++ and managed CLI code within the same assembly (.dll or .exe file).

A major disadvantage of C++/CLI is that there exists an implementation of it only for the Windows platform.

3
/ C++/CLI library containing both
// unmanaged C++ and managed C++/CLI
#pragma managed(push, off)
int add(int a, int b)
{
    return a + b;
}
#pragma managed(pop)

namespace CppCliMath
{
    public ref class Math
    {
        public:
            static int Add(int a, int b)
            {
                return add(a, b);
            }
    }
}

// C# consumer
using System;
using CppCliMath;

class Program
{
    public static void Main()
    {
        Console.WriteLine(CppCliMath.Add(117, 132));
    }
}
**Platform Invoke** Platform Invoke (PInvoke) is a Common Language Runtime (CLR) feature that supports calling exported functions in unmanaged dynamically linked libraries.[5] The Mono .NET framework also supports calling Unix shared libraries (.so files).

Unlike C++/CLI, PInvoke method requires that the unmanaged part of the code be compiled into a separate library file.

```csharp
// C++/C library math.dll
int add(int a, int b)
{
    return a + b;
}
```

```csharp
// C# wrapper
using System;
using System.Runtime.InteropServices;

class Program
{
    public static int Add(int a, int b)
    {
        return add(a, b);
    }

    [DllImport("math.dll", EntryPoint = "add")]
    private static extern int add(int a, int b);
}
```

```csharp
// C# consumer
using System;
using System.Runtime.InteropServices;

class Program
{
    public static void Main()
    {
        Console.WriteLine(add(117, 132));
    }

    [DllImport("math.dll", EntryPoint = "add")]
    private static extern int add(int a, int b);
}
```
1.4 Binary interface of DLL files

PInvoke relies on the application binary interface (ABI) of dynamically linked libraries (DLL files). There is a standard platform and compiler independent ABI for C-style functions, but C++-style functions use compiler-specific name mangling and therefore cannot be relied on for interoperation.

This means that wrapping C++ classes using PInvoke requires writing extension methods that translate methods calls from C++ to C. Three levels of indirection are required to support this type of interoperation.

<table>
<thead>
<tr>
<th>Method</th>
<th>Exported name (MSVC)</th>
<th>Exported name (GCC 4)</th>
</tr>
</thead>
</table>
| // C++-style method class Math
| { public:
| static
| int add(int a, int b);
| } | ?add@Math@@SAHHH@Z | ZN4Math3addEii |
| // C-style method int math_add(int a, int b); | math_add | math_add |
| // C extension method // for C++ method int math_add(Math* obj,
| int a, int b) |
| { obj->add(a, b); } | math_add | math_add |

2 Mapping of language features between C++ and C#

For interoperation, we need to list language features in C++ and C# and define a mapping between them. Since we are only interested in calling functions, it is enough to consider features used in function declarations. These include namespaces, visibility specifiers (public, private), function declarations and types. Types include basic types (int, float), classes, structs and enums.
2.1 Basic types

When translating method signatures from C++ to C#, basic types of method parameters can simply be renamed (e.g. `void m(unsigned int a) → void M(uint a)`).

Built-in types in C# are aliases of types in the .NET System namespace.[6] C++ method names which include type names should be renamed to use the .NET framework type names (e.g. `read_short()` → `ReadInt16()`). This is necessary because the methods could be called from other .NET languages where types have differing aliases (for example, in Visual Basic, a boolean is named “Boolean”, but in C# it is named “bool”). The .NET framework provides a unified naming scheme for such built-in types.

<table>
<thead>
<tr>
<th>C++ type</th>
<th>Corresponding C# type</th>
<th>.NET framework type</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long</td>
<td>ulong</td>
<td>ULong</td>
</tr>
<tr>
<td>unsigned int</td>
<td>uint</td>
<td>UInt32</td>
</tr>
<tr>
<td>unsigned short</td>
<td>ushort</td>
<td>UInt16</td>
</tr>
<tr>
<td>unsigned char</td>
<td>byte</td>
<td>Char</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>Int64</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>Int32</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>Int16</td>
</tr>
<tr>
<td>char</td>
<td>sbyte</td>
<td>SByte</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>Boolean</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>Single</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>Double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++ method signature</th>
<th>Corresponding C# signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>void m(unsigned long a);</td>
<td>void m(ulong a);</td>
</tr>
<tr>
<td>unsigned int read_uint();</td>
<td>uint ReadUInt32();</td>
</tr>
</tbody>
</table>

2.2 Classes and access methods

Access methods (getter and setter pairs) can be represented with C# properties.

```csharp
// C++ class with getter & setter
class Math
{
    int getAlpha();
    void setAlpha(int a);
}
```
// C extension methods for the C++ class
Math* math_new() {
    return new Math();
}

int math_getAlpha(Math* o) {
    return o->getAlpha();
}

void math_setAlpha(Math* o, int a) {
    o->setAlpha(a);
}

// .NET wrapper class
class Math
{
    IntPtr _native;
    public Math()
    {
        _native = math_new();
    }

    public int Alpha // property
    {
        get { return math_getAlpha(_native); }
            set { math_setAlpha(_native, value); }
    }

    [DllImport("math.dll", EntryPoint = "math_new")]
    static extern IntPtr math_new();
    [DllImport("math.dll", EntryPoint = "math_getDelta")]
    static extern int math_getAlpha(IntPtr o);
    [DllImport("math.dll", EntryPoint = "math_setDelta")]
    static extern void math_setAlpha(IntPtr o, int a);
}
2.3 Fields

For some libraries, it may be necessary to provide access to fields. Fields can be wrapped using a pair of get/set methods, which can in turn be mapped into a C# property.

<table>
<thead>
<tr>
<th>C++ feature</th>
<th>Corresponding wrapper code</th>
</tr>
</thead>
<tbody>
<tr>
<td>// C++ field</td>
<td>// C extension methods:</td>
</tr>
<tr>
<td>int m_delta = 5;</td>
<td>// field getter &amp; setter</td>
</tr>
<tr>
<td></td>
<td>int getDelta()</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>return m_delta;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>int setDelta(int delta)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>m_delta = delta;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>// .NET wrapper library</td>
</tr>
<tr>
<td></td>
<td>// C# property</td>
</tr>
<tr>
<td></td>
<td>public static int Delta</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>get { return getDelta(); }</td>
</tr>
<tr>
<td></td>
<td>set { setDelta(value); }</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>[DllImport(&quot;math.dll&quot;, EntryPoint = &quot;getDelta&quot;)]</td>
<td>private static extern int getDelta();</td>
</tr>
<tr>
<td>[DllImport(&quot;math.dll&quot;, EntryPoint = &quot;setDelta&quot;)]</td>
<td>private static extern void setDelta(int delta);</td>
</tr>
</tbody>
</table>
2.4 Resource cleanup

When a consumer has finished using a resource, memory for it has to be reclaimed. In general, C++ uses deterministic cleanup, meaning that memory has to be freed explicitly by the programmer. On the other hand, .NET uses non-deterministic cleanup, meaning memory is reclaimed when the Garbage Collector (GC) determines that the object is no longer in use.

.NET classes that hold unmanaged resources (e.g. pointers to C++ object instances), must implement the IDisposable interface. The class will then have a callback method called Dispose, which is responsible for freeing the unmanaged resource. The Dispose method can be called either explicitly by the programmer or implicitly by the GC when the object goes out of scope.

```c++
// C++ class
class Clock {
   long getUnixTime();
}

// C extension methods for the C++ class
Clock* clock_new() {
   return new Clock();
}

void clock_free(Clock* o) {
   delete o;
}

long clock_getUnixTime(Clock* o) {
   return o->getUnixTime();
}

// .NET wrapper class
class Clock : IDisposable {
   IntPtr _native;
   public Clock() {
      _native = clock_new();
   }

   public long GetUnixTime() {
      get { return clock_getAlpha(_native); } 
   }
}
```
// .NET wrapper class (continued)
public void Dispose()
{
    Dispose(true); // Dispose called explicitly
}

~Clock()
{
    // GC called the destructor,
    // now call the Dispose method
    Dispose(false);
}

protected virtual void Dispose(bool userDispose)
{
    // Free unmanaged resource
    clock_free(_native);
}

[DllImport("clock.dll", EntryPoint = "clock_new")]
static extern IntPtr clock_new();

[DllImport("clock.dll", EntryPoint = "clock_free")]
static extern void clock_free(IntPtr o);

[DllImport("clock.dll", EntryPoint = "clock_getUnixTime")]
static extern long clock_getUnixTime(IntPtr o);

// C# consumer
using System;

class Program
{
    public static void Main()
    {
        Clock clock = new Clock();
        Console.WriteLine(clock.GetUnixTime());
        clock.Dispose();
    }
}

In case the programmer forgets to call clock.Dispose() explicitly, the memory will still be freed thanks to implicit garbage collection.
3  Mapping of coding conventions between C++ and C#

There are several conventions used in C++ code for symbol naming, code formatting and scoping. These are enforced not by the language specification, but by some organization. For example, Google uses its own style guide.[8] Microsoft provides its own guidelines for .NET.[2] Since library consumers expect code to adhere to such conventions, then the conventions must be changed during conversion from C++ to C#.

3.1 Capitalization

<table>
<thead>
<tr>
<th>Google C++ Style Guide</th>
<th>Corresponding C# naming convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>// constant</td>
<td>// constant</td>
</tr>
<tr>
<td>const int kDaysInAWeek = 7;</td>
<td>const int DaysInAWeek = 7;</td>
</tr>
<tr>
<td>// variables</td>
<td>// variables</td>
</tr>
<tr>
<td>int price_count_reader;</td>
<td>int priceCountReader;</td>
</tr>
<tr>
<td>int tablename;</td>
<td>int tableName;</td>
</tr>
<tr>
<td>// private variable</td>
<td>// private variable</td>
</tr>
<tr>
<td>private:</td>
<td>private int <em>numEntries</em>;</td>
</tr>
<tr>
<td>int num_entries_;</td>
<td></td>
</tr>
<tr>
<td>// namespace</td>
<td>// namespace</td>
</tr>
<tr>
<td>namespace bullet</td>
<td>namespace Bullet</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

3.2 Prefixes

Some C++ libraries (such as the Bullet physics library) use prefixes instead of namespaces to group classes. Since .NET does not allow prefixes by convention and requires classes to be placed into namespaces, it is reasonable to remove the prefixes and convert them into namespaces instead.

<table>
<thead>
<tr>
<th>Bullet physics library</th>
<th>.NET (C#) wrapper for the library</th>
</tr>
</thead>
<tbody>
<tr>
<td>class btCollisionObject</td>
<td>namespace Bullet</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>};</td>
<td>}</td>
</tr>
<tr>
<td>class btRigidBody</td>
<td>class RigidBody</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>};</td>
<td>}</td>
</tr>
</tbody>
</table>
3.3 Inline documentation

Both C++ and C# support inline documentation for methods. Doxygen is a common tool and specification for generating documentation based on annotations in C++ comments. Doxygen also supports C#, but Microsoft’s own feature for this is called “XML Documentation Comments”. The two approaches share similar annotations, but have different syntax. Automatic conversion is straightforward.

<table>
<thead>
<tr>
<th>C++ Doxygen documentation</th>
<th>C# XML Documentation Comments</th>
</tr>
</thead>
</table>
| /*!
  * \brief Addition.
  * \details Adds two numbers.
  * \param a first term
  * \param b second term
  * \returns a+b */
| int add(int a, int b) {
  return a + b;
}; | int Add(int a, int b) {
  return a + b;
}; |

4 Automated wrapper library creation

Given the rules above, the conversion process can be automated. The first step is to create the abstract syntax tree (AST) of the C++ header files. The relevant information is extracted from the AST and stored into a custom in-memory representation. Next, rules are applied to convert class structures and symbol names from C++ style to .NET style. Project-specific rules are then applied to add custom behavior or to handle complicated cases that could not be handled automatically. Finally, the new code files are written to disk together with project files for Visual Studio or other build environments.

4.1 AST parsing

An AST can be generated using libraries such as Clang, GCC-XML or ANTLR. We will be using Clang as it is better maintained than GCC-XML and more specialized than ANTLR.[7] The generated AST can then be traversed to find all the relevant information such as namespaces, classes, method declarations and fields.

Clang allows method bodies to be skipped during code traversal. This can speed up the parsing process if there happen to be method bodies inside include files (.h). Methods definitions are usually written inside code files (.cpp), which are excluded anyway.

The implementation of the above is hosted at: https://github.com/AndresTraks/BulletSharpPInvoke/tree/master/BulletSharpGen
5 Future work

There are more C++ language features that could be wrapped for use in C#, but some of them have complications. For example, template classes can be very difficult to wrap, because the wrapper class requires a reference to a concrete C++ class. In other words, a C++ template class such as Vector<T> cannot be wrapped using a C# class Vector<T>. Rather, a concrete wrapper class (Vector3) can wrap a single specialization of the class (Vector3). This situation can potentially be solved by determining which specializations of the template class (Vector2, Vector3, etc) are used in the library and creating a wrapper class for each one (Vector2, Vector3).

The performance differences between C++/CLI and Platform Invoke could also be analysed. C++/CLI is potentially faster thanks to linking to unmanaged code directly, while PInvoke requires multiple layers of indirection.

The above approach has only been applied to the Bullet physics engine, but using it on more libraries should reveal what potential shortcomings there might be for general use.

6 Glossary

.NET .NET framework, a software framework for .NET languages such as C#

CIL Common Intermediate Language, an intermediate representation of program code used by the .NET framework

CLI Common Language Infrastructure

C++/CLI C++ on Common Language Infrastructure, a language designed for C#-C++ interoperation

Managed code Code that is compiled into CIL bytecode, e.g. C#

Native code Platform-specific code that is compiled into machine code directly, e.g. C++

PInvoke Platform Invoke, an interoperation method

Unmanaged code Native code
References