Graph Rewriters as Components

Using a GROOVE-based API in selected use cases

Dennis Aanstoot
mail@dennisaanstoot.nl

August 25, 2022

Summary

This thesis describes an API for a general purpose graph transformation tool set that uses Graph Rewriting, a technique where simple labeled graphs are transformed using transformation rules. The aim of the study is to make the Graph Rewriter interoperable with other system, so these other system can benefit from the complex logic the tool set provides. This study finds requirements for the API and creates an API design that meets those requirements. Multiple designs and approaches for APIs are discussed, after which one design is developed.

In order to verify the newly created API, three use cases are worked out to check out the usefulness of the API. The use cases are different in nature, to test the usefulness in different scenarios. The first use case is a chess engine, the second is a processor simulator and the third and last one is LEGO Mindstorms.

On the basis of the findings from these use cases conclusions are drawn.

Contents

1 Introduction

1.1 Motivation

Technology has entered our lives. We cannot imagine life without it. Computers are in our homes and our workplaces, and powerful mobile phones are in our pockets. These products are modern pieces of engineering. They have improved over the years due to the progress that has been made on two aspects of technology:

- The hardware side of the product, the physical electronic parts
- The software, like applications and operating systems, which run on the hardware

The development of software uses techniques to improve the speed of producing software and increase the complexity of software. New techniques are discovered regularly and they are used to improve the development process of software.

One of the techniques for improving the software development process is code reuse. When writing software, some of the code does not have to be rewritten if it is already publicly available. This saves time because one does not need to write this code themselves. This improves both development speed and code simplicity. When code is reused on a running system, the code only has to be loaded once, and can be used by multiple process instances at the same time. On most computer systems some kind of code reuse is utilized. There are multiple ways to reuse code.

1.2 Interoperability

One of those ways of code reuse is the simplest: copy all files that contain the needed code to your project and use that code. This will work, but this has disadvantages. Any bug that the original developer resolves in the code or any extension they make, after you copied it, will not be applied to your software project. You would have to keep track of the upstream code for changes, or find and solve found bugs yourself. Moreover, if more software projects make use of the same copied code, this would mean you have to copy the same code over many projects, and update the copied code on each project that uses it. This increases the amount of work that has to be done, and increases the chance of a human error.

A more modern way of reusing code is by making use of an Application Programming Interface (API). The API is a description for software that describes how the software can be used by external code, so how a program can be interoperable with another program. This description tells the programmer how they can use the functionality the software behind the API provides.

There are multiple kinds of APIs, which will be discussed in Chapter 2, but what they have in common is that code is deferred to a new project into projects with different purposes, increasing the separation of concerns, and projects can be reused by making use of the API. The API and the code it uses, become its own software project, which releases new versions when improvements have been made. This can be added as a dependency in projects which use the API, without code copying.

In Figure 1 a sequence diagram is shown where an application makes use of an API that handles all mathematics the application needs to calculate. The API defines how an answer to a mathematical problem should be asked from the library. In this case, we can see that *sqrt* is a known keyword for this API, and it takes the square root of the expression between the parentheses.

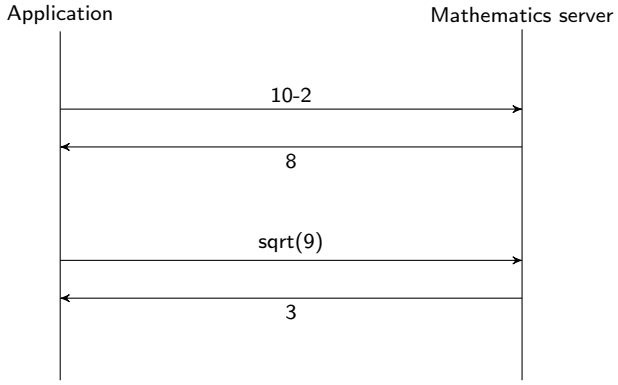

Figure 1: Making use of an API.

There are several ways of using an API. This fictional Mathematics service is used by sending request messages to it and receiving a response message for each request. This is how web-based APIs work. A real-world example for this is the API for Wikipedia, the online encyclopedia. We can open the following URL in a browser: `https://en.wikipedia.org/w/api.php?action=query&format=json&generator=search&gsrlimit=3&gsrsearch='Netherlands'`. It would give us the following result:


```
{
  "batchcomplete": "",
  "continue": {
    "gsroffset": 3,
    "continue": "gsroffset||"
  },
  "query": {
    "pages": {
      "21148": {
        "pageid": 21148,
        "ns": 0,
        "title": "Netherlands",
```

```
        "index": 1
      },
      "80482": {
        "pageid": 80482,
        "ns": 0,
        "title": "Beatrix of the Netherlands",
        "index": 3
      },
      "18949613": {
        "pageid": 18949613,
        "ns": 0,
        "title": "Kingdom of the Netherlands",
        "index": 2
      }
    }
  }
}
```


The URL consists of a part that defines where and how to connect (`https://en.wikipedia.org/w/api.php`), and a part after a question mark that contains the message for the request.

- `'action=query'` makes the call a query
- `'format=json'` defines the format to response should be in
- `'generator=search'` defines that we want to search
- `'gsrlimit=3'` limits our search to three results
- `'gsrsearch='Netherlands''` defines the search term to use

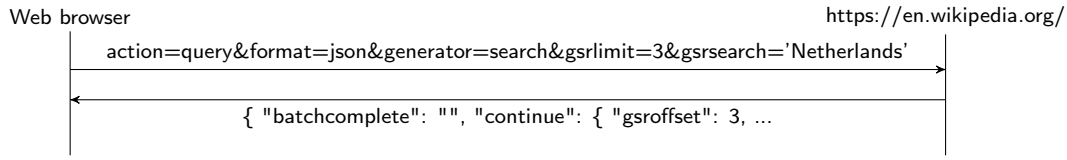

Figure 2: Making use of the Wikipedia API.

This example shows a lot of parameters and the response message is quite complex, but the exact meaning of all elements is not important here. It is important to notice that this is a method for programs to exchange information with each other. The explanation of what parameters are possible and what the result should be is all part of the API. An application that wants to couple itself to Wikipedia, can use this API to execute its Wikipedia searches with.

Now we take a look at a real-world use case for using an API that does not use messages, but method calls. A method is a code block that contains a

series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. The API describes which method calls are possible, and what result the user can expect from each call.

Say we want to write a program to play music and we want the program to be able to play MP3 files. MP3 is a file encoding which compresses audio files. Because the technical details of the MP3 encoding are quite complex, it is hard and error prone to rewrite the algorithms for the encoding ourselves. Fortunately, the LAME project[8] provides MP3 logic. This software library can be installed on most operating systems, and when we write our music program using the LAME API, we can use the logic LAME provides, which is tested to correctly implement the MP3 encoding.

Additionally, if LAME releases a new version of the library, and the API has not changed, we can benefit from this by just updating LAME. The code for our MP3 player program does not have to change to use all the improvements made to the LAME library. The improvements can vary from bug fixes to performance improvements.

Figure 3 shows a sequence diagram of how the API would work in an MP3 player application.

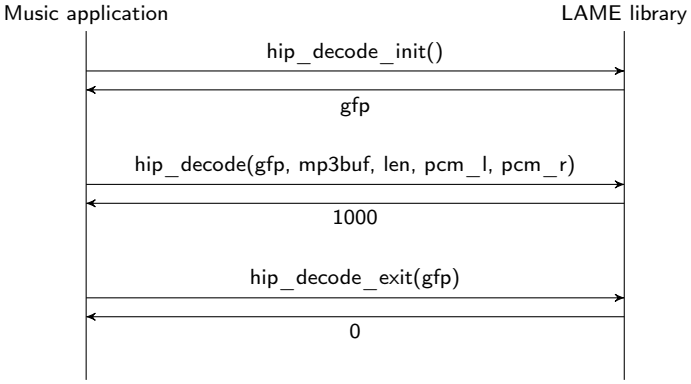

Figure 3: Using the LAME library.

In this example the LAME library is used to decode an MP3 stream, which means translating the MP3 to something an audio device can understand. In the first call, `hip_decode_init`, the library is started, and `gfp` (a data structure to configure the decoder) is the result of the call. In the next call, the library is called to do the actual decode work. It requires arguments:

- `gfp`, the config we got from the last call
- A pointer `mp3buf` to where the MP3 can be found
- The `len`(gth) of the MP3 buffer

- Two buffers (`pcm_l` and `pcm_r`) where the audio result for the left and right audio device can be stored

The result of the value of the call is a number, expressing the samples worth of data the call was able to decode. In the last call the decoder is closed. It uses `gfp`, the result we got from the first call. The API in this case describes each method we used here. The description for `hip_decode` is illustrated in Figure 4. Note that for this example not all text here is important. For a developer it is. It describes how the API behaves.

```
/*********************************************************************
 * input 1 mp3 frame, output (maybe) pcm data.
 *
 *  nout = hip_decode(hip, mp3buf,len,pcm_l,pcm_r);
 *
 * input:
 *     len            :  number of bytes of mp3 data in mp3buf
 *     mp3buf[len]    :  mp3 data to be decoded
 *
 * output:
 *     nout:  -1      : decoding error
 *             0      : need more data before we can complete the decode
 *            >0      : returned 'nout' samples worth of data in pcm_l,pcm_r
 *     pcm_l[nout]    : left channel data
 *     pcm_r[nout]    : right channel data
 *
 *********************************************************************/
int CDECL hip_decode( hip_t             gfp
                    , unsigned char *   mp3buf
                    , size_t            len
                    , short             pcm_l[]
                    , short             pcm_r[]
                    );
```

Figure 4: Method description for hip_decode.

1.3 Graph transformation

GROOVE[6] is a tool developed at the University of Twente. It uses Graph Grammars to generate a transition system consisting of graphs as states and partial graph morphisms as graph transformations. Before we can take a look at what the software does, to give an intuition we need to take a look at graphs and graph transformation.

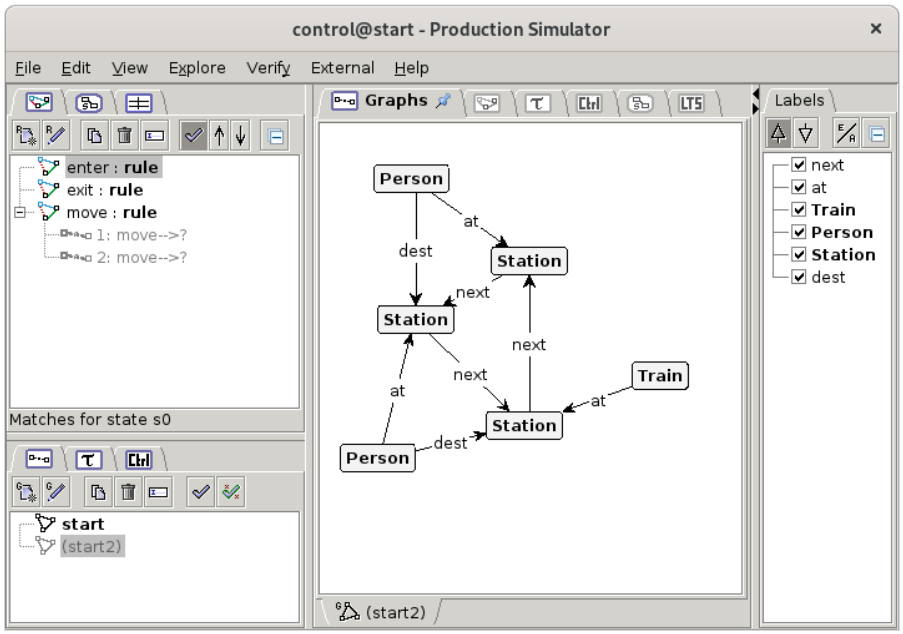

Figure 5: Screenshot of GROOVE.

A graph is a mathematical model that is used to model objects and relations between them. A graph consists of a set of nodes and a set of edges. Nodes are often represented using circles or rectangles, and edges are represented using lines, connecting two nodes with each other. Nodes and edges both can have labels. In Figure 6 a graph is shown with four nodes and five edges.

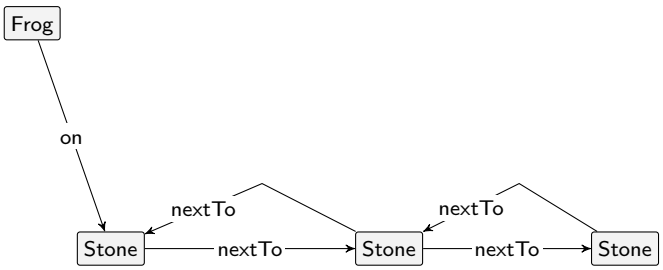

Figure 6: A graph modeling a frog and stones.

This graph models a frog and three stones. The frog and stones are represented as nodes in this graph. The frog has an edge to a stone, with the label `on`, describing for this example that the frog is located on a stone. The pairs of stones have a directed edge with the label `nextTo`, describing that these stones are adjacent.

Graphs are often used to describe real-world situations/problems. In the example in Figure 6 for example describes a frog in a pond with some stones.

The graph in Figure 5 shows a train, some stations and some persons willing to go from one station to another. So-called graph transformations rules are used to describe changes to these situations by changing the original graph to a new one. The transformation rule consists of a graph L and a graph R. If a morphism exists of graph L to graph G, this transformation can be applied to graph G. In that case the image of L in G will be replaced by R, and graph G will be transformed. In the case of the graph in Figure 5 for example, graph transformations cam be used to move the train from station to station.

The tool GROOVE is a computer program that can simulate graph transformations on labeled graphs. It has a GUI (Graphical user interface) where graphs and graph transformations can be constructed and the result of transformations can be viewed. In GROOVE one or more starting graphs and a set of graph transformation are combined in a Grammar. Figure 5 shows GROOVE with a Grammar loaded.

GROOVE currently does not have a well-defined API, so other programs cannot make use of the functionality GROOVE provides. If this would be available, programs can interoperate with GROOVE, and by doing this use the logic GROOVE provides. This has as prerequisite that the program that wants to use GROOVE benefits from this interoperability. This requires that these programs do something similar that can be replaces by logic provided by GROOVE. If this is the case, those programs can replace there own logic with a connection to GROOVE which will do this logic for them. This can simplify the code for the program that makes use of the GROOVE API.

In this project an API for GROOVE is created to make it interoperable with other programs. If GROOVE has an API, programs that make use or could make use of graph transformations, would be able to reuse the logic of GROOVE. A choice has to be made between different API types and it has to be decided which components the API exposes. After the API is created, it will be validated with use cases that confirm its usefulness.

1.4 Objective/Validation

To validate the usage of the API, the GROOVE API is connected with already existing software projects. In those projects, pieces of code that can be described using graphs and graph transformation are removed, and replaced with a connection to GROOVE. The use cases show whether GROOVE can be effectively used in the code.

This makes two benefits for every use case:

1. With code reuse in mind, it would be an improvement if this simplifies or decreases the amount the code. Grammars will now contain the logic that was previously written in code.
2. The GROOVE connection can now be loaded with slightly different grammars and this will result in different behavior in the overall system without

changing any programming code.

1.5 Report structure

The rest of the report has the following structure:

- Chapter 2 contains background information about graphs, graph transformations and APIs
- Chapter 3 describes the design of the API for GROOVE
- Chapter 4 describes multiple use cases which will validate the usefulness of the GROOVE API and lists the results of the use cases
- Chapter 5 summarizes the findings and draws conclusions

2 Background

This section contains information about GROOVE, web technology and APIs. All information has been researched by other people, and is publicly available for anybody to use and base their own work on.

2.1 GROOVE

GROOVE[6] is a program that uses graph grammars to generate a transition system consisting of graphs as states and partial graph morphisms as graph transformations. Before we can take a look at GROOVE, we need to understand what graphs and graph transformations are.

Graphs

In Figure 7, a graph is shown which models two frogs and three stones. The graph is a little different from the graph in Figure 6. A fly has appeared on the right most stone. The frog and the fly both have an edge to a stone with the label `on`, describing that they are located on that stone.

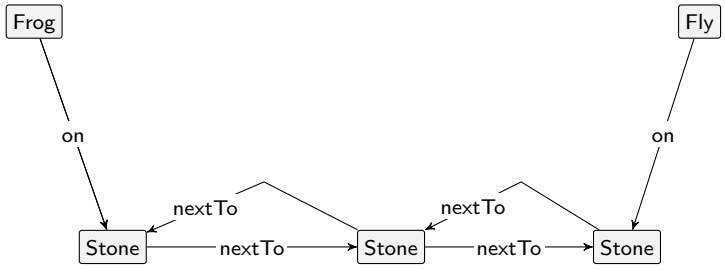

Figure 7: Graph G modeling a frog on a stone.

Note that in this graph the edges have a direction, expressed using an arrow. This is opposed to graphs where graphs are expressed as lines, meaning that the edges do not have a direction (undirected graphs). This research only uses directed graphs.

Graph transformations

The graph in Figure 7 is modeling a problem. The frog wants to eat the fly, but first it has to get to the same stone as the fly first. The frog can jump one stone if the stone is adjacent. Stones are adjacent if there is a `nextTo` edge to the next stone it. We can describe the frogs jumping using a graph transformation rule. This would make the frog able to move.

With this rule we can apply graph transformations[19], which takes a graph as input, and outputs a new graph after performing this rule. To capture the transformation, we need 2 graphs. One that models the state before the rule is performed and one which models the state after the rule is applied.

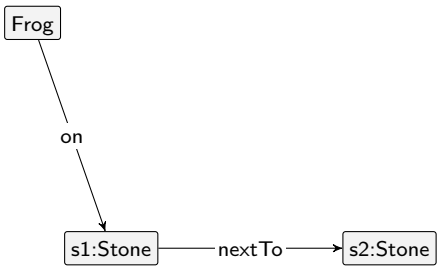

Figure 8: Graph L for the first graph transformation rule.

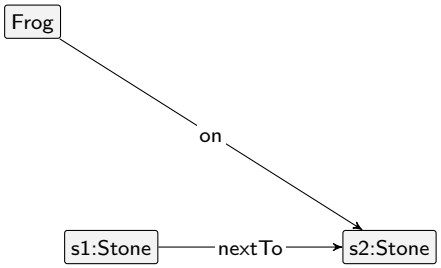

Figure 9: Graph R for the first graph transformation rule.

Graph L in Figure 8 shows a graph that describes a start situation where our frog can jump from one stone to the next, the pre-condition. The frog is on a stone and there is a stone adjacent to the stone where the frog is on. The extra labels s1 and s2 are used to identify the two stones in Graph R. s1 and s2 represent the same stones in both L and R. In Figure 10 the image of L is highlighted to visualize the morphism.

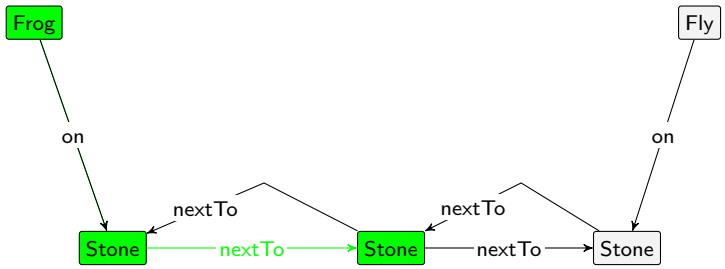

Figure 10: A graph G with L highlighted

Graph R in Figure 9 shows the situation after a frog has jumped a stone, the post-condition. The frog has jumped from one stone s1, followed the directional

`nextTo` edge, and is now on the stone the edge is pointing to. The s1 and s2 indicate that these are the same nodes in L.

A transformation can take place if an injective morphism exists between Graph L and the Graph G we want to transform. A morphism exist between Graph L and Graph G if a function exists for the set of nodes of L to the set of nodes of G that respects the structures of the graphs. In our example a morphism exists for graph L in graph G. There is one occurrence of this morphism in G, so we can do the transformation.

Performing the transformation intuitively takes the following steps:

1. Find an occurrence of L in the given graph G.
2. Delete from G all vertices and edges matched by $L \backslash R$ in the occurrence.
3. Paste to the result a copy of $R \backslash L$, yielding the derived graph G'.

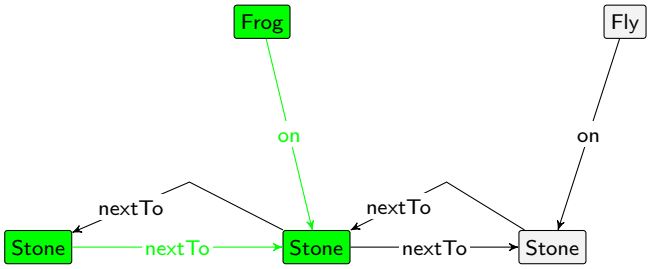

Figure 11: A graph G', the transformed graph.

Figure 11 shows G', after the transformation. The frog moved from the left most stone to the middle stone. In this figure the post-condition R is highlighted.

Note that in G' L occurs twice, so if we want to apply the rule again there we can apply the rule in two ways. One where the frog jumps to the left, which when performed would result in graph G again. The other occurrence is where the frog will jump to the right. Both possibilities are shown in Figure 12. We want to get to the fly, so we apply the transformation that will take us to the right:

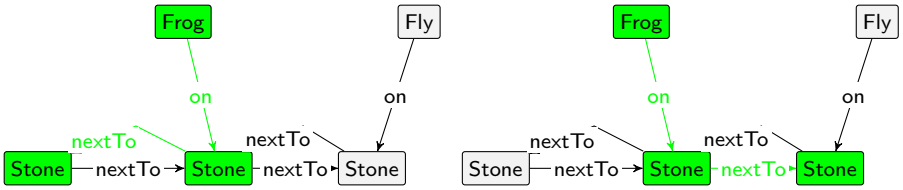

Figure 12: Graph G' has two possible transformations.

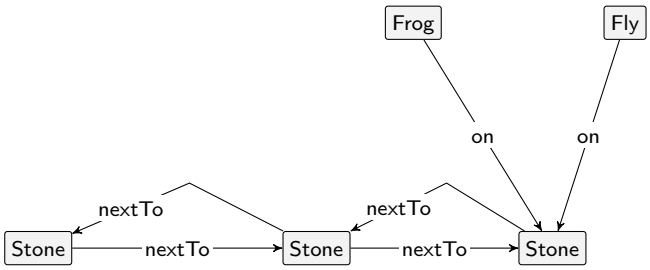

Figure 13: The frog and fly on the same stone.

Now that the frog has reached the fly, it can eat it. A second rule is needed that can delete the fly if the frog is on the same stone. This rule again consists of two graphs: a graph L, the pre-condition, as shown in Figure 14, and a graph R, the post-condition, as shown in Figure 15. Graph L shows that the frog and the fly should be on the same stone. This ensures this rule cannot be applied in the starting graph from Figure 7. The post-condition does not contain the fly. The fly node will be deleted if this rule is applied.

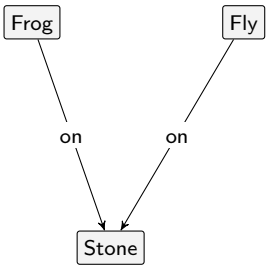

Figure 14: Graph L for the second graph transformation rule.

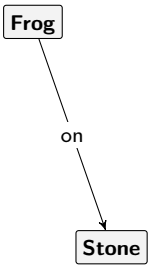

Figure 15: Graph R for the second graph transformation rule.

Applying the second rule will result in the final state for our problem, as seen in Figure 16. The fly has disappeared from the graph. The frog was able to eat the fly:

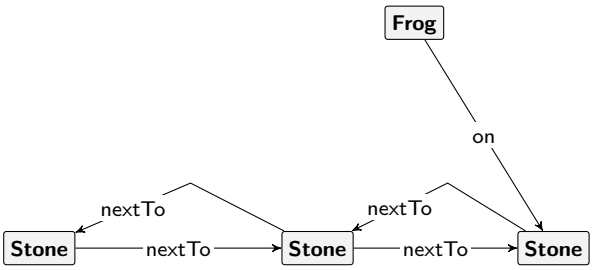

Figure 16: The frog has eaten the fly.

GROOVE

GROOVE is a tool developed at the University of Twente that is able to model graphs and simulate graph transformations. The Simulator program of GROOVE provides the user with a graphical user interface.

All components that are needed for graph transformations are bundled in a Graph Grammar. A grammar consists of:

- A Start graph is the begin situation on which further transformations can be performed on. The start graph consists of nodes and edges. Nodes and edges optionally have labels to describe additional data.
- A Type graph defines the type structure the start graph should comply with. A type graph defines what types are possible, which edges are possible between types, and it supports a subtyping system which can define type hierarchies. The type graph is optional in a grammar, but it can prevent the grammar creator from making mistakes in start graphs and rules. Figure 17 shows a Type graph for the Frog example.
- Rules are graphs that show a possible graph transformation. In GROOVE specific, rules consist of only one graph instead of two, but by using keywords the user can create context over what the transformation changes about a graph. For example, the keyword 'new' will create new elements, and the keyword 'del' will delete elements after a transformation. GROOVE supports keywords that change the behavior of the transformation. Figure 18 shows the 'jump' rule, which makes the frog move from one stone to a second. All nodes and edges except for the green edge need to exist in a graph for a transformation to be possible. Is a transformation is performed, the blue edge is removed, and the green edge is added to the new graph.
- Control scripts give more control over the flow of the possible transformations. The scripts are text files that will run from top to bottom, and declares the possible next transformations for the current graph.

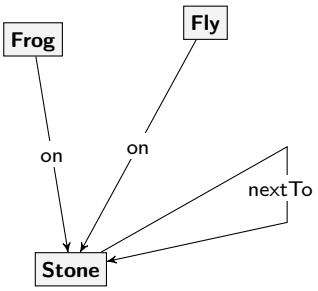

Figure 17: Type graph for the Frog example.

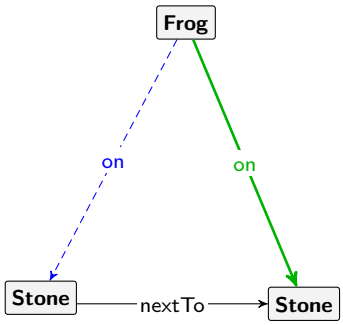

Figure 18: The 'jump' rule for the Frog example.

GROOVE uses these grammars to store information to simulate graph transformations. GROOVE can use a start graph, and select a possible transformation, and apply it. GROOVE will generate a Labeled Transition System (LTS), which is a graph that depicts for each node a graph state, and for each edge a transformation. The starting node of the LTS is representing the starting graph(s), and for each possible transformation a new edge points to a new node, that depicts the new graph after the transformation, unless the same graph has already been found. Then the edge will point to the node for that graph. With the example of the frogs this means that the first node of the LTS would represent graph G. This node would have an edge describing the rule of the frog moving to the target node that represents graph G'.

Figure 19 shows the LTS that is generated for the example where the Frog eat the Fly. The start graph is represented by 's0'. After two jumps we can get to 's2'. There we can do an 'eat' transformation. After that we cannot get back to 's0'. We can only 'jump' back and forth.

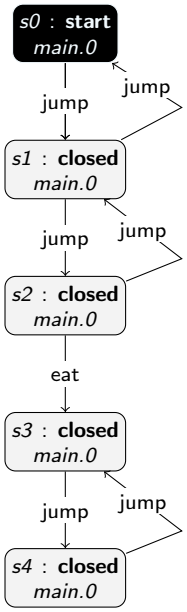

Figure 19: The LTS of the Frog example.

2.2 JSON

Data interchange formats are designed and used when data has to be sent over a network; although, they are also used to store data on a storage device. The formats are designed to be readable by machines and by humans. All general interchange formats are able to contain the same data, only in a different structure. The differences are mainly syntactical. The choice for one of these formats is mostly a matter of taste and preference of the developer. The most common formats are XML[2] and JSON[15]. To explain data interchange formats, we will take a look at JSON. An example of a state representation of a car is given in Figure 20.

```
{
    "make":"VW",
    "model":"Polo",
    "wheels":4,
    "rimsizes":[15,15,17,17]
}
```

Figure 20: A car, represented in JSON.

JSON is a syntax for a data structure that supports multiple data types. The root of the structure is called a JSON Object, which is a data type in itself. The JSON Object consists of key-value pairs, separated by commas, and

surrounded by curly brackets. The keys are always of data type string. The values can be of any type.

Besides this JSON Object data type, there are more types. The raw data types in JSON are:

- String
- Number
- Boolean
- null

Additionally, there are types that allow embedding and summing of types. These are Sets, which are comma-separated elements surrounded by curly brackets, Lists, which are comma-separated elements surrounded by square brackets, and the JSON Object again, which can be used in other places as the root of the structure.

These are the basic features of JSON. Extensions exist that give more possibilities to express data. However, in most applications this default set of features is used.

2.3 TCP

Applications can sent their data over a network channel. A channel is a tool to send messages to and receive messages from other users. It is used to connect two programs, so that data can be interchanged. It is used to connect two computers via a network connection. In that case, both computers run a program that use the channel to communicate. Although the name network channel suggests that connection should be between two computers, it is also possible to connect two programs that are running on the same machine.

TCP (Transmission Control Protocol)[12] is a protocol for the Transport Layer of the OSI model, and takes care of reliable transportation of data through a network. Often TCP is used when using a network channel that needs reliability. TCP has features that makes the connection more reliable than other alternatives such as UDP[11] (User Datagram Protocol), which is an alternative protocol where packets of data can be lost during transit. All messages that are sent are guaranteed to be delivered in TCP, and are received in order, as long as the connection persists. This is resending packets that are not delivered correctly. Moreover, the TCP protocol has mechanisms to let the program know that the other side of the connection is not reachable anymore, in other words, when the connection has been lost.

We can send any message over a TCP connection as long as it is expressed in bytes. APIs can be written by writing a custom application layer protocol on top of TCP to complete the OSI model. It is also possible to use an application level that has been predefined by others, like for example HTTP.

2.4 HTTP

When the features of TCP are not enough, one could decide to use Hypertext Transfer Protocol (HTTP)[18] on top of TCP. Additional elements of this protocol are, the identification of web resources using URIs (uniform resource identifiers), separation of request methods, and the requirement of a response message.

HTTP is a network protocol between a client and a server that opens a new connection for each new request, and closes the connection after a response. Each HTTP request consists of at least a request method and a URL.

The request method is one of the possible requests from the set of possibilities described in [18]. The request method describes the purpose of the request. Some examples of request methods are GET, PUT, DELETE and POST, which are used to respectively receive a resource, edit a resource, delete a resource, or create a new resource. All request methods have properties they need to cohere to, like the GET request method is Safe, which means it will never change data on the server. A PUT is Idempotent, which means it can be repeated, but the result after one call is the same as after two calls. In other words, if these two are the only requests incoming, the second request will do nothing.

In addition to this, a URL is added, which is a concatenation of:

- A scheme, like "http://"
- A domain (sometimes called 'authority'), like "www.example.com"
- A path, like "/car/1"

The complete URL would be "http://www.example.com/car/1". The protocol string describes how to connect, the domain which server to connect to, and the URI the resource we are interested in. The URL can be extended to add more context, but this is the simplest form. URI is the way for web technologies to identify different resources on a server. A resource can be an object, such as people and places, a concept, or an information resource such as a web page or a book. The URI is represented as a multiple strings, concatenated with slashes ("/"). In our example we are looking for a car resource with identification number 1.

When trying to send data regarding a resource, a URL and a request method are not enough. In this case a request body has to be added. It is the agreed way to attach data to a HTTP request. Often a machine and human readable form is used, like JSON or XML. When expecting data from the response, it can be found in the response body.

After a request from a client to a server, the server will sent a response to the client. The response will contain a response code. It is a three-digit number describing the attempt to understand and satisfy the request. For example, 200 means the result has been found and is contained in the response. 404 means the

result could not be found, and that the response will not contain the requested data. After each request follows a response. A TCP connection between the client and the server is opened before sending a request, and is closed after sending a response.

Sessions are tools in HTTP to store properties for a specific client. This is mainly used to authenticate a client, but it also enable other uses, like storing additional context about the connection. Often cookies are used to maintain sessions. Cookies are pieces of data that are unique for a client, and contain information about the client. The cookie is stored by the client, and is sent along with each request. Sometimes the cookie only contains a unique hash that identifies the client. In this case the server can store client information based on this identifier. It is also possible to store information about the connection client-side in the cookie, so the server does not have to store this.

2.5 API

There are multiple ways to separate the logic of a program, and contain it in a distinct project. One of the prominent ways to do that is by exposing the important logic as an application programming interface (API).

The two types of APIs that are important to us are Software libraries and Web APIs. The first imports logic together with the program using it. The second exposes itself as a program which an open network channel, and any actor can connect and make use of this library. In the next two subsections we are going to take a look at how both techniques work, and look at the advantages and disadvantages of both techniques.

2.5.1 Software library (API)

Let's take a look in the case where the API is exposed as a software library. The precise execution of this concept differs for different programming languages and different platforms, but some things are in common. The software library is exposed as a package containing (often precompiled) code. During runtime, the precompiled code gets embedded in the main program when the main program runs. This is an automatic process, which makes a software library easier to set up than the alternative Web API, which requires additional actions at runtime. Loading the library in memory also has the consequence that the library code runs on the same machine as the main program code.

During the development of a program that uses a software library, we need to know how to use it. A software library has a set of components, of which a subset are exposed. Often these components are method declarations or abstractions of method declarations with classes or namespaces. A method is a block of code which only runs when it is called and it results in a return value. We can pass data, known as parameters, into a method, which will provide context to the method. An API would describe for each method call what the methods and

its parameters are representing, how data would be changed afterwards, and what the return value of the method call would be. This API can be used by software developers that want to make use of the software library. The exposed components can be accessed by an external program.

A common disadvantage of software libraries is that they put limits on the choices of software developers. Most software libraries require the program using them to be written in the same (or a similar) programming language. As an example, when a software library is written in Java it should be loaded into the Java Virtual Machine (JVM) for it to be usable. The program using the software library therefore also has to make use of JVM and thus it should be written in a language that can compile to an object language that the JVM understands. For our example, this would mean that the program using the software library should be written in languages such as Java, Scala or Kotlin. Languages such as Python or C# are not viable options as they do not produce object code for the JVM.

Workarounds exist to connect software libraries from different languages. For Java libraries it is possible to connect it to the C programming language by using JNI[3]. However, this will give C programs that use a Java library a lot of overhead because it has to load and configure and start the JVM to be able to use the library.

Advantages

- Easy to startup at runtime

Disadvantages

- Requires library running on same machine as main program
- The number of programming languages that can use the library is limited

2.5.2 Network API

Network APIs can be divided in stateless and stateful APIs. The differences are mainly about whether the state of the program are stored. This subsection will explain the distinctions between them.

Stateless API

Stateless protocols have become popular these days. Many APIs are REST-based. REST[17] (REpresentational State Transfer) is an example for explaining what statelessness means.

REST is a software architecture style which defines how requests should be using a uniform and predefined set of stateless operations. The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves. A service that is compliant with the principles of REST is called RESTful.

In REST, if a request or response contains data for a resource, it will be the complete state. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any context stored on the server. This is the core of what stateless means for a REST service. The service could be stopped and started again without clients experiencing any difference.

REST is language- and protocol-independent, but always requires an underlying application layer protocol to process requests. HTTP is often the underlying protocol used.

Using HTTP, the representation of the car from Figure 20 could be sent to a fictional car server using a POST request on the "/car" URI, after which the server adds this new car to its data storage, and sends a response with a new car ID to the client. Requesting this new ID from "/car/{id}", where {id} is replaced with the new car ID, should return the VW Polo from the server. The car would be available to any client who requests it, because the server is not aware of any sessions, because that would involve extra context stored in a state, so it ca not distinguish between clients.

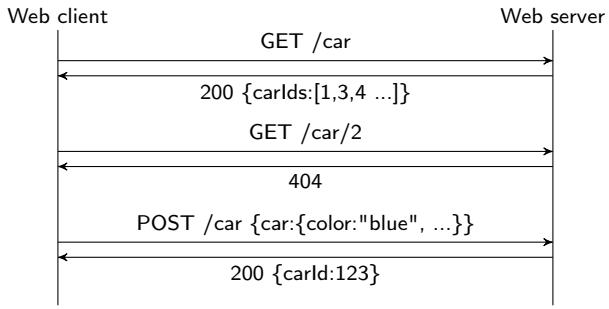

Figure 21: Using a RESTful server for cars.

In Figure 21 a possible interaction with the car server is shown. The calls are hypothetical calls of how the server could be designed.

- The first call is a GET call and the response consists of the IDs of all stored cars. The response status code is 200 (OK).
- The second call tries to GET one specific car with the ID 2. The server

responses with 404 (Not found). The car with ID 2 does not exist.

- The third call stores a new car with a POST call. The call includes JSON data with data about the car, like for example that it is a blue car. The response has a 200 status code (OK), and a JSON response is included with the new ID for the new car.

Stateful API

Some protocols have a more stateful design. Examples of these protocols are FTP[10], SSH[22] and Telnet[13]. Note that the names 'protocol' and 'API' are used here. These two terms are closely related. APIs are used for programming purposes, and protocols are a set of rules to communicate are protocols APIs allow two applications, sub-systems, etc, to communicate and possibly control each other. Sometimes protocols are used for that. An example would be to establish a GIT connection using the SSH protocol. The SSH protocol explains the language SSH communicates with. The API is still a GIT API, and SSH is the protocol. When SSH is used as an interface to program against, it is used as an API. We can show what a stateful design is on the basis of what FTP offers.

FTP is a file sharing protocol allowing a client to connect to a server, traverse through the directory structure and retrieve files from or send files to the current directory. It is important to note that FTP is directly built on top of TCP and thus does not use an application layer protocol as opposed to REST. TCP is used because the reliable connection it provides. Moreover, the connection will be kept alive for as long as the connection is required. Also a disruption of the connection will destroy the state. FTP, Telnet and SSH all make use of TCP for these reasons.

The FTP protocol consists of a list of known commands terminated by the ASCII character sequence CRLF which are sent by the client through the TCP channel, and the server will respond to each command with the data requested, or some error code.

Some of the commands FTP supports are:

- LIST to list the files in a directory
- RETR to retrieve a file from the server
- STOR to store a file on the server

The FTP program is able to store session-specific information about the user in memory. When the connection is ended or disrupted for some reason, the program can dispose this session data.

Stateful protocols are extendable in the way that commands that are defined, should be supported in newer versions to enable backwards compatibility, but new commands can be added when new functionality is required.

Another interpretation of a stateful protocol is the RPC[24] (Remote Procedure Call) protocol. It is a protocol used to connect a client to a server, but abstraction makes sure that for the client, there is not much difference compared to doing the procedure calls locally. These types of protocols use frameworks to make sure both the client and the server communicate correctly and without errors.

RPC has more than one implementation, some are language specific, some not. For Java, the language specific JRMI[4] exists, which is included in the default Java Runtime. Using JRMI starts with constructing a Java Interface, which defines which methods will be exposed using the connections. An example can be seen in in Figure 22, which exposes the method 'power'.

```
public interface RmiServerInt extends Remote {
    BigInteger power(int a, int b)
            throws RemoteException;
}
```

Figure 22: JRMI interface

Using this interface a server can be constructed that exposes an instance of class to the network. This can be seen in Figure 23. The Server class implements the interface RmiServerInt from Figure 22 extends a class to make it suitable for the usage of JRMI. The main method exposes this specific implementation of the interface to "//localhost/Server".

```
public class Server extends UnicastRemoteObject
        implements RmiServerInt {
    public Server() throws RemoteException {
        super(0);
    }

    @Override
    public BigInteger power(int a, int b) {
        return BigInteger.valueOf(a).pow(b);
    }

    public static void main(String[] args) throws Exception {
        try {
            LocateRegistry.createRegistry(1099);
        } catch (RemoteException e) {
            System.exit(-1);
        }
        Server server = new Server();
        Naming.rebind("//localhost/Server", server);
    }
}
```

Figure 23: JRMI server

When the server is running, a client can connect to the server and make procedure calls there. The client does need to have access to the same JRMI Interface the server is using (the one from Figure 22). In Figure 24 a main method connects to the same name the server exposes to.

```
public class Client {
    public static void main(String args[]) throws Exception {
        Remote lookup = Naming.lookup("//localhost/Server");
        RmiServerInt serverProxy = (RmiServerInt) lookup;
        System.out.println(serverProxy.power(2, 32));
    }
}
```

Figure 24: JRMI client

When the Client runs this main method, the object 'serverProxy' will function like a proxy. Every call that is done on it, will be redirected to the server with the implementation that is exposed there. For the developer there is no difference in programming after the object has been created by connecting to the server. The object is in reality living in the memory of the server, and will stay there as long as the server is exposing this object.

Using this structure of calling this remote object, the server is able to hold the state for the object, and will even keep the state after the client disconnects. The object is not user or connection private, but rather shared with all connections. Multiple clients can connect to the same object and change the state together. This can be solved by making the proxy object have a 'login' design pattern, a method which creates a new (remote) object. This structure ensures that every client uses its own unique object.

This sample with the Java JRMI implementation is an example of how RPC works, and is also an example that has a big flaw: the implementation is language specific. Protocols exist that are language independent. Two examples of a protocols that are actively used are SOAP[16] and gRPC (`https://www.grpc.io`).

Summary

Now that we understand how a program can communicate with a Network API, we can summarize the advantages and disadvantages.

Because the library and main program are running as separate programs, and the connection is network-based, the programs are able to run on two different machines, which makes the structure of the program more dynamic. The Network API program would support more than one user to run on it, and data could be shared by multiple actors. Also the computing power of multiple computers is utilized, which is a plus.

The structure of having separate programs has the disadvantage that the user needs to know how to use both programs. They both have to be started, and they have to be configured to connect to each other.

Advantages

- API can be running on a different machines
- Every programming language that supports network connections can be used

Disadvantages

- Takes more preparation to run compared to a software library

3 API Design

This section covers the decisions that were made for connecting to the new GROOVE API. This section contains the scope of what features of GROOVE the API supports, and shows the technical details of the API, and the requirements that resulted in this design.

3.1 Scope of API functionality

GROOVE has multiple features that could be supported by the API, but for this project the simulating of existing Grammars will be supported. Creating, editing and exporting of Grammars is not in scope for this project. The use cases care about simulating existing grammars only. Creating and editing Grammars using an API whould be a whole project on itself.

3.1.1 Requirements for the API

The API is should cover the following requirements:

1. The API should be language independent
2. The API must be able to load a Grammar
3. The API must be able to get possible transitions for a state
4. The API must be able to traverse states in the LTS
5. The API must be able to return to earlier visited states
6. The API must be able to get node and edge information of a state
7. The API should be able to get all existing rules in a Grammar
8. The API should be able to get all existing types in a Grammar

3.2 Choice for GROOVE API

For the GROOVE API a Network API is chosen, and not a software library solution. The reason for this is to satisfy the first requirement, to make the API language independent. Software libraries have the disadvantage that only comparable languages can make use of the library. By creating a Network API all languages that support network protocols can use the API. Moreover, the API will not be stateless because of some complications with such a design.

3.2.1 Problems with Stateless

In this project we build a Network-based API, because of the advantages this type of API brings. An advantage of using a Network API is that the user can use any programming language. It covers the first requirement: "The API should be programming language independent". A software library is language dependent, so this is not an option. For the second requirement: "GROOVE is a computation heavy program, so the API should not lead to performance loss", a stateful design is easier to configure without wasting resources.

Although a stateless Network API would meet our requirement, trying to design an API will uncover problems that will make making use of such an API problematic. We will illustrate these problems using examples.

A characteristic of REST is that the requests that are available to a client do not depend on the requests the client has done in the past, because that would involve session context. The response of a request can however change over time, because the data storage of the server can be changed by clients.

Let's say we have a stateless GROOVE server. In order for a GROOVE server to be stateless, the API should be defined. A grammar with its rules can be represented as a (JSON) state, and can be saved as such in the server, at the URI "/grammar" for example. We would now be able to lookup the starting graph of the LTS on "/grammar/{id}/lts/0", which would return the possible state IDs which are reachable using the applicable rules. We need to keep in mind that states in the LTS can be heavy to calculate. There are some questions to be answered beforehand. Are these resources available for all clients on the server? If so, how long are these resources available? If this grammar is only available for one user, then we need to record a session for this client, and everything this client sees is hidden for other clients, which makes the stateless property of the server redundant.

The architectural model of REST does not match with the usage of cookies[17]. The cookie adds additional context to each request, while the purpose of REST is to not have context, just a state. So the usage of sessions in a RESTful service is ruled out. REST luckily is not a doctrine that we need to follow by the line, so we could accept a session structure in our application, and focus on other concepts of REST.

Independent of the choice of using sessions, we get to a caching problem. Because connections to the server are ad-hoc on a request basis, and a clear exit message of a client to the server does not exist in stateless protocols, we do not know for how long the server should store its data. A choice would be to not dispose data without a user requesting a deletion, but this would put much responsibility on a user that is using the API in an ad-hoc way, and data would be stored indefinitely. To fix this, a cache approach is possible, where after an amount of time, the data is disposed, which would be unfortunate if the user accesses a disposed resource, and the process of calculating graph transitions has to start over, which is processor intensive. This would violate the second requirement.

3.2.2 Stateful

We can work around the problems we have with stateless API designs by using a stateful API. For a mature API a protocol can be beneficial because of better code maintainability, more compact code, and a smaller change on making mistakes as the framework's boilerplate will push the developer in the right direction. For now we are creating a prototype, so we define our own custom protocol that suits our requirements.

The connections starts with creating a TCP connection. Because we do not want the problems with performance and the burden of caching, we do not want the underlying TCP connection to close (like HTTP would do after each response), so we will keep it alive for as long as required. For the duration of the connection, we keep all data we need in memory, and when the connection closes we can drop the data. Both requirements for the GROOVE API can be satisfied with a stateful design.

Instead of exposed objects or methods like in RPC, we expose a set of 'commands' to users of the API to use. The commands respond with data about running simulations, like graph information or state information.

3.3 Commands for GROOVE

The main way the stateful API communicates is by sending commands, which can have zero or more parameters. A command is as a string representation over the TCP connection, and is ended with a newline character ('\n'). After a command is received by the API server, a response is sent as a string, mostly in the JSON format, depending on the command type. A response is also ended with a newline character. Because all commands will be custom, we need to create a list of commands that would be required to reach our goals for this project.

The commands that are implemented are:

1. LOAD, the command to load a grammar
2. RULES, the command to receive the rules in the loaded grammar
3. TYPES, the command to receive the types defined in the loaded grammar
4. STATE, the command to receive the structure of the graph in a given state
5. INFO, the command to receive the possible transformations for a given state
6. MATCH, the command to feed a state with ask parameters
7. EXIT, the command to close the connection

The LOAD command will be accompanied with a model for a grammar as an parameter for GROOVE to load. The model is a zipped GROOVE Grammar, encoded in base64[20]. Base64 is used because the set of possible characters it uses is limited, and the endline character '\n' is not used. After this command is used and a grammar is loaded, the other commands are available to get information about the grammar from the API. The response of the command is "DONE" if the grammar is successfully loaded.

The RULES command returns a list of all rules that are described in the grammar. This command does not have any parameters. The response for the command is a JSON Object. A grammar should be loaded for this command to work.

The TYPES command returns a list of types described in the Type graph of the grammar. This command does not have any parameters. The response for the command is a JSON Object. A grammar should be loaded for this command to work.

When a grammar is loaded, we need commands to start the simulations. A command to get information about the states in the LTS (Labeled Transition System). In the LTS all nodes have an identifier (e.g. s1, s2, s3, etc.), which can be used to identify a specific graph.

The STATE command is implemented to get the graph information about a specific node in the LTS and a single parameter, the state number of the node. The API responds with a graph representation in a JSON Object. This is a representation for all nodes and edges in that state.

Of a state in the LTS an INFO command with a state number as parameter would give us the possible transformations that are possible in this state. A possible transformation would consist of at least a rule name and a target state.

GROOVE has a mechanism called an Oracle. This means that when a parameter is unknown at the time of running the simulation, the Oracle will determine the value of the parameter. For example the Default Oracle will give it a default value (0 for a integer), and a Random Oracle will generate a random value. A special oracle is the Dialog Oracle, which in the GROOVE GUI interface program will present the user with an input field, where the user can provide the program with an input for the unknown parameter.

The MATCH command is available when the Dialog Oracle is used, and uncompleted transformations exist, to complete missing parameters for the transformations. The missing parameter is added to the request, where-after it is sent via the network connections. This commands needs two arguments: a state number, and a list of values for incomplete parameters.

The INFO command will, besides normal transformations, also show transformations where parameters are incomplete. Using the MATCH command the Dialog Oracle could be provided with information about the missing parameter.

The EXIT command does not need a parameter and is simple, it just closes

the connection and discards all data.

The API is backwards compatible, because adding new functionality by introducing new commands will not change existing commands. As long as the changes to the API consist of adding new commands, current commands will function as intended. Figure 25 shows a summary of all commands.

command	arguments	description	when available?
load	[.gps.zip file in base64 encoding]	Before anything can be done, a grammar has to be loaded. The grammar is a GROOVE grammar (.gps folder), which is zipped and encoded in base64. If the grammar is loaded, this command will respond with "DONE\n", or "ERROR\n" when something goes wrong. If a grammar is already loaded, this command overwrites that grammar.	Anytime
rules	-	This returns the a JSON list of all rules in the loaded grammar	When a grammar is loaded
types	-	This returns the a JSON list of all types that exist in the type graph of the loaded grammar and the primitive types that GROOVE supports.	When a grammar is loaded
state	[statenumber]	When a grammar is loaded, this command returns the graph for a given state number. The starting state has number 0. Any other numbers become available when the info command has discovered them.	When a grammar is loaded
info	[statenumber]	When a grammar is loaded, this command returns a list of possible transitions in the given state. The stating state has number 0. Any other numbers become available when the info command has discovered them.	When a grammar is loaded
match	[statenumber] [list of parameters]	When a grammar is loaded, and the state has an incomplete transformation because of an Oracle value to be provided by the Dialog Oracle, this command completes that transformation.	When a grammar is loaded
exit	-	Just exits	Anytime

Figure 25: Table of possible commands in GROOVE API

4 Validation

In this chapter, we report on three use cases intended to show the benefits and drawbacks of using GROOVE for the part of a program that can be expressed by graphs and graph transformations. This means that parts of the program's will be deferred to GROOVE, and hopefully simplify the program. An additional benefit is that the grammar loaded in GROOVE can be exchanged with a grammar that will give the program different behavior, without any code to be changed.

These three use cases are selected because they are different in nature and show different aspects of usefulness of the API.

These use cases will provide the validation whether the GROOVE API is useful, and can be used in different projects. The use cases will be checked against the points made in section 1.4. The requirements listed in section 3.1.1 will be checked for completeness.

4.1 Giraffe (Chess engine)

This subsection covers the use case where a Chess engine gets coupled with the GROOVE API. Moreover, the chess engine will be changed in such a way it can also play connect four, nim and tic tac toe due to the interchanging of grammars for these games.

4.1.1 The project

Giraffe[21] is an open source chess engine which is developed by Matthew Lai at Imperial College London. It uses deep learning in combination with self play to train a neural network in playing chess with minimal hand-crafted knowledge. It keeps track of a tree with possible states for its minimax algorithm.

Because it originally is a project for chess, some chess specific pieces need to be changed to be more generic. These things are:

- The notation for encoding piece positions.
- The Feature set for training the neural network.

FEN

The original Giraffe implementation uses FEN[9] (Forsyth-Edwards Notation) to encode piece positions on a board. An example of a FEN is:

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq – 0 1

The notation consists of all rows, top till bottom, separated by slashes ('/'). The number '8' means the row is empty. In addition to the rows, some additional information is stored. These are:

1. The color who is next to do a move. (w)
2. The available castling rights. (KQkq)
3. The (optional) position where en passante is possible. Can be no position. (-)
4. The number of half turn since the last capture, also called a ply. (0)
5. The number of full turn, so a turn for both players, so two plies. (1)

This FEN example represents the starting state of a chess game. There are two problems with this notation.

1. The notation is chess specific, and is not suited to encode boards from other games.
2. Because every board has to be constructed using transformations, additional code has to be written to convert a board to transformations, which will construct the intended board configuration.

Both problems can are solved by replacing this chess specific notation with a GROOVE specific one.

Feature set

Giraffe uses a feature set to encode the entire board into a single representation. It is a set of features of the pieces and other properties of the current board that can be fed to the neural network as representation of the board. As stated in [21]: the features should be of low enough level and be general enough that most of the chess knowledge is still discovered through learning. The main issue is that we need to remove chess specifics from the feature set, which might be essential for the neural network to perform optimally.

If one would really want to add game specific features to the neural network, it would be possible to extend Giraffe with the possibility to read these values from the Grammar. Specific nodes in the graph could function as input for the training set. For this prototype this has not been implemented.

4.1.2 Connection with GROOVE

The use case is to replace the chess logic by a GROOVE Grammar. At this moment Giraffe has its own code to calculate all possible moves in chess. This

part of the code has not anything to do with the machine learning logic. Because the powerful part of the software is the machine learning, and the logic for the generation of possible moves is just necessary for it to function with chess, we can replace this logic with a GROOVE API connection. In the new situation, the minimax algorithm gets information about possible moves from a Grammar that is running in GROOVE.

The starting graph for chess is the starting board, and each possible move, corresponding to a certain chess piece type, can be modeled as a graph transformation. This way all rules of chess can be modeled. In the GROOVE Grammar, each node in the LTS would represent a chess board configuration, and each edge leaving the node is a possible move by a piece on the board. We will store the state space of the games of chess that are played on the Giraffe client.

Giraffe will be stripped of all of its chess logic. Instead, this will be replaced with a GROOVE API connection and a GROOVE Grammar. Using this connection, Giraffe can ask for possible moves given the current state. Giraffe needs to keep track of some state information, but state numbers are sufficient. It needs the numbers to identify the states when Giraffe is building a search tree to find the best move.

When Giraffe uses the GROOVE API connection, we can do something that would not be possible with chess logic written in Giraffe's codebase. We could try to feed GROOVE with a GROOVE Grammar for another but similar game. If we made a grammar for, for example, connect four, we would expect Giraffe to perform its usual tree search. With the new grammar loaded the connection returns connect four board states, and the possible moves on the connect four board. This would require no changes to the programming of Giraffe whatsoever. With different grammars, Giraffe would be able to learn itself to play any board game.

4.1.3 Use case expectations

We consider this use case to be a success if:

1. Giraffe can be adapted to make use of a connection to GROOVE to find possible chess moves.
2. The chess grammar can be changed to a grammar for another board game, and Giraffe would be able to learn itself that game.

For the following games GROOVE Grammars were made and tested:

- Chess
- Connect Four
- Nim
- Tic-tac-toe

4.1.4 Implementation

Giraffe is a chess engine written in C++. The logic for chess is an integral part of the program. This part of the code is replaced with a connection to the GROOVE API. Because Giraffe needs to be able to undo moves, a stack of traversed state numbers is stored, and undoing a move is as easy as popping the top element of the stack.

The FEN notation is replaced with a more abstract one. The notation is the list of transformations needed to construct the board. This can currently be done in two ways.

The first way is by using a list of 'set' transformations. This 'set' rule has two parameters. The first is the piece type and color, represented as an integer, and the second parameter is a position from 0 to 63 for the places on the board. 'set' transitions are always applied on an empty board because of the flow of the control script.

The second way is by performing a list of 'move' transformations on a normal starting board for chess. Sometimes this is easier to store than generating 'set' transformations from a board. To distinguish this notation from the list of 'set' transformations in an early stage of parsing, this notation starts with the '#' symbol.

The feature set for a GROOVE powered Giraffe is way simpler than the original Giraffe, but nevertheless, a neural network can be created by running the learning steps provided in the README of Giraffe. Because the feature set is more abstract, the network should need more iterations to learn specific about games.

Giraffe has a default Feature set with the castling rights for both players, and attack and defend maps, which contain the lowest-valued attacker and defender of each square. These are too specific for chess, and are removed. What remains are the color who's turn it is, the number of pieces for each type for both colors and the piece lists. These features are generic enough to work for any board game that meets a number of requirements:

1. The board's maximal size is a eight-by-eight rectangular grid
2. The game is turn-based
3. The game has two players
4. No random factors (like dice rolls)
5. The control script is the same script as shown in Figure 29

The maximum board size is more a requirement for the internal method that prints the board to the standard output, so this requirement is not really necessary for the program to function, but printing the board would not work

as expected. The TUI could in theory be altered to contain easily accessible information about the dimensions of the board, but for this use case that would create unnecessary complexity. The game should be turn-based and a two player game, for the minimax algorithm to still function. Also, a random factor is not implemented in Giraffe, the outcome of a move is always the same if repeated.

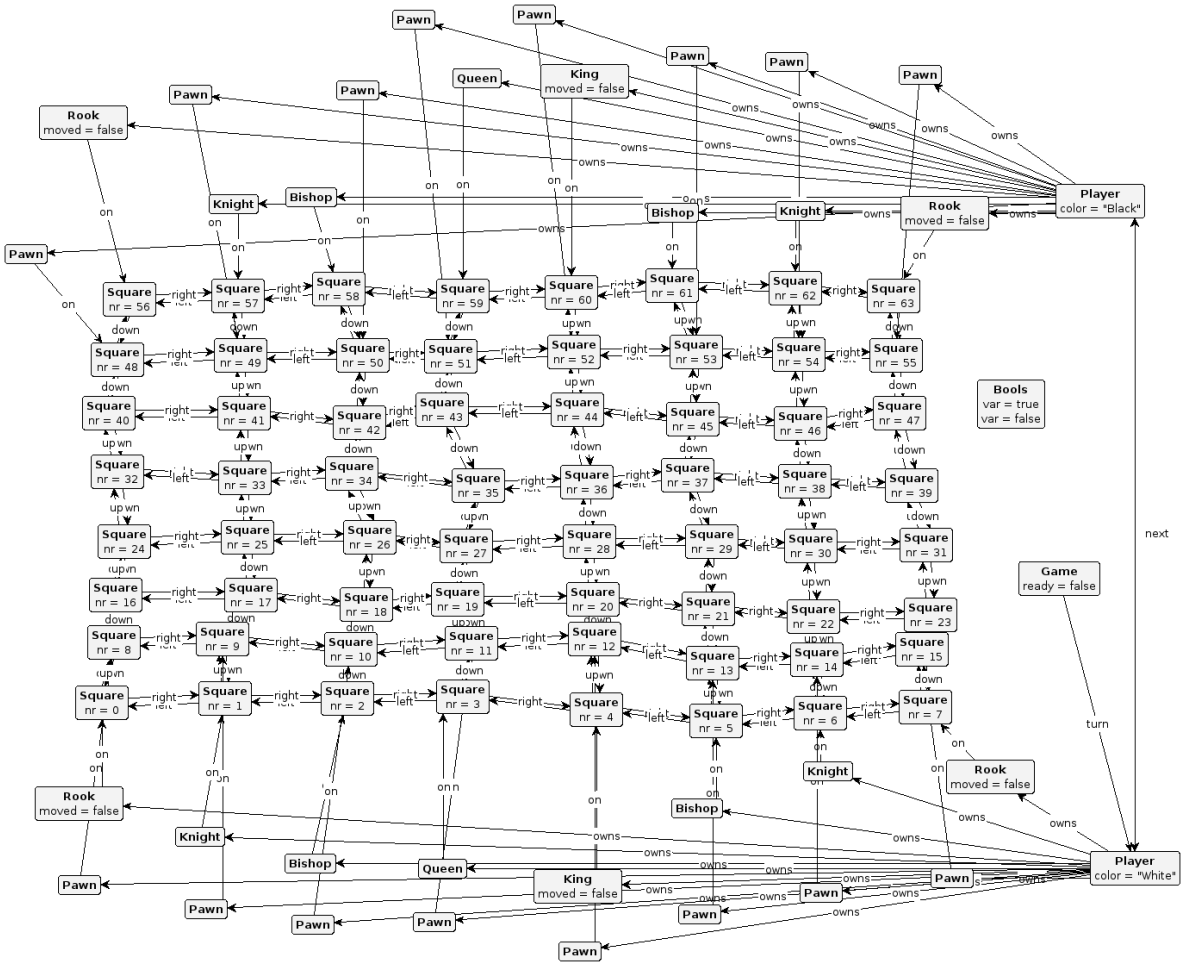

Figure 26: Starting graph for chess.

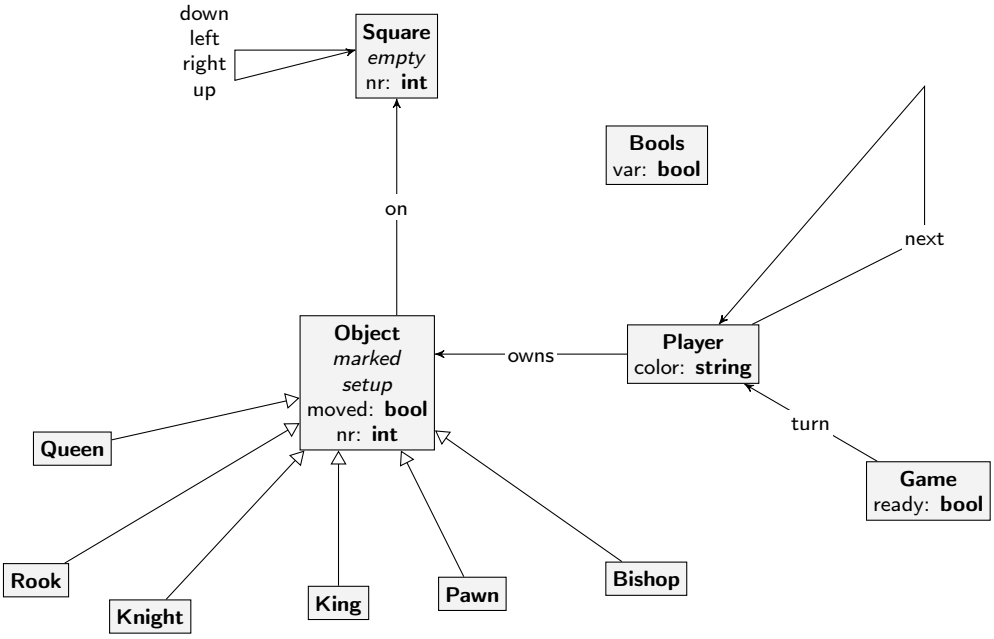

Figure 27: The Type graph for chess.

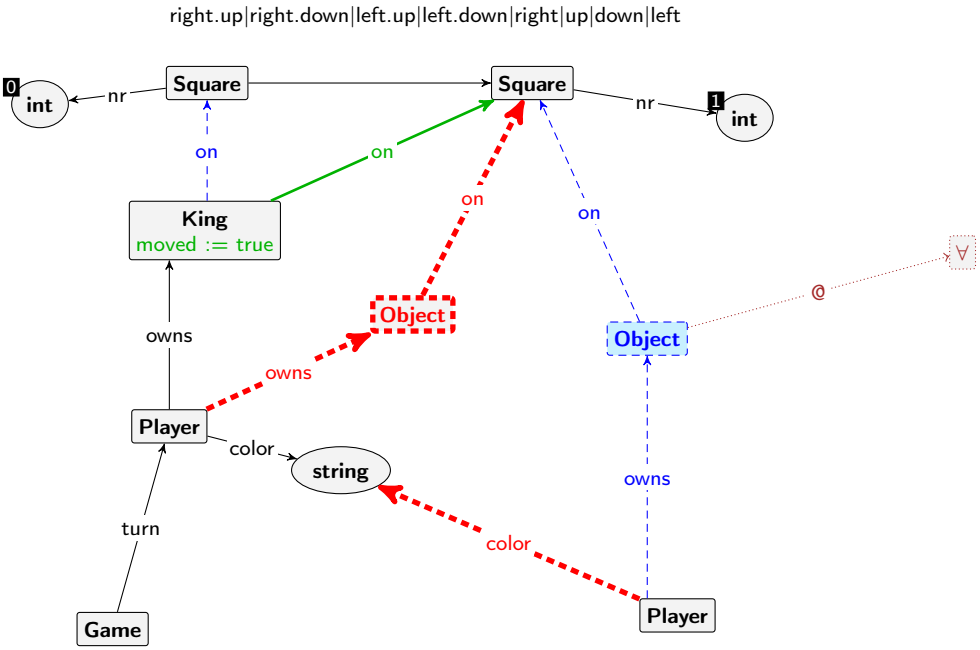

Figure 28: The 'moveKing' rule for chess.

The GROOVE Grammar contains a start state of a chess board with all pieces at their starting positions (Figure 26). Every rule represents a move

that a piece type can make. The GROOVE Control script has a structure that should be compatible with board games of a similar design. When the Grammar is replaced by a different one, with the same rule names and Control script, the Grammar should work with Giraffe. This also means that in the case of chess, every piece type specific moves, and all special moves like castling, are reduced to a simple move rule. The type graph can be found in Figure 27. Figure 28 shows what a move of a king piece looks like. A king can move according to the regular expression '(right.up)|(right.down)|(left.up)|(left.down)|right|up|down|left', and the new spot should not have an own piece on it. The pieces of the opponent are removed from the new spot if they exist.

Figure 29 shows a snippet of the GROOVE control code that is used for the chess grammar. The script starts with a rule called "start". This sets a global variable which is used by "notReady". When "start(false)" is called, "notReady" will be true. The block in the if statements clears the board, and gives the user the possibility to place pieces on the board to create a custom board with the "set" rule. This can be useful for when the learn algorithm of Giraffe needs a specific board. When the board is ready, custom or not, "refreshSquares" is used for some initialization of the board. In the chess grammar this sets flags on all squares that do not contain a piece, which are necessary for the move rules to work.

```
bool d;
start(out d);

if (notReady) {
  empty();
  while (notReady) {
    int a;
    int b;
    set(out a, out b);
    bool c;
    start(out c);
  }
}
refreshSquares();
while (notEndGame) {
  int a;
  int b;
  move(out a, out b);
  nextPlayer();
}

recipe move(out int a, out int b) {
  choice moveRook(out a, out b);
    or movePawn1(out a, out b);
    or movePawn2(out a, out b);
    or movePawn3(out a, out b);
    or movePawn4(out a, out b);
    or movePawn5(out a, out b);
    or movePawn6(out a, out b);
    or movePawn7(out a, out b);
    or movePawn8(out a, out b);
    or moveKnight(out a, out b);
    or moveKing(out a, out b);
    or moveQueen(out a, out b);
    or moveBishop(out a, out b);
    or castle1(out a, out b);
    or castle2(out a, out b);
  refreshSquares();
}
```

Figure 29: Snippet of the control script for the Chess Grammar

Using this structure makes the control script usable for a variety of board games. The “move” makes it possible to do an abstract move without context about the game or existing piece types.

With the chess grammar created, it is time to make it work with Giraffe. There are two move evaluator types in Giraffe: static and neural network based. The current implementation for the neural network move evaluator is too chess specific, so some changes have been made here to support more types of games. Changing the static move evaluator to be move generic and still expecting it to function well, is somewhat impossible, because the main part about a static evaluator is that there are programmable best moves for a game, because the game has been analyzed by the programmer. However, changing the neural network evaluator to be more general/generic might work, although the loss of some chess specifics will probably increase the number of learning iterations needed to learn a game, because all removed game specifics have to be learned by the neural network instead.

The speed of iterations is way slower, and GROOVE is using a lot of memory to process all move possibilities. The neural network that is produced, is able to play a game of chess, but has not learned a lot. The moves are basically random. Giving Giraffe more time to do more iterations of learning results in GROOVE going out of memory, because the state space that is explored is too big. This means Giraffe has become unable to learn enough to qualify this part of the use case a success when considering the game of chess.

Giraffe's code uses caches to mitigate memory problems, but the caches of GROOVE seem to be insufficient for this scale of different states. Because chess is a game that has shown to have a state space that is too big, a smaller game will be tried. Connect four is a game that meets all requirements listed above. Moreover, the state space is more compact compared to chess.

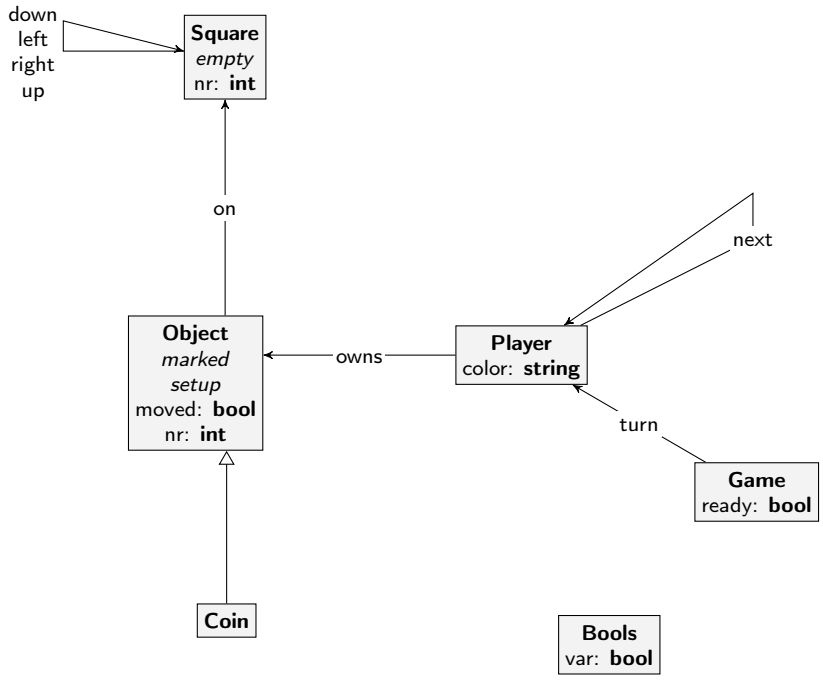

Figure 30: The Type graph for connect four.

The starting graph of connect four is the same as that of chess (Figure 26), except it only contains Squares, Players, Game and Bools. The type graph is similar but simpler, as it only has coins, and no other types that can exist on the board (Figure 30)

Even though connect four seems to be a simpler game with a smaller state space, it is still too big for GROOVE to handle. The state space for a board with a seven-by-six grid is 4531985219092 as shown on `https://oeis.org/A212693`. The equation to solve this is basically $7^{7\times6}$, but it takes into account that nearing the end of the game some slots are not available anymore, because they have been filled up till the top. Trying a Grammar that simulates connect four will result in GROOVE going out-of-memory very soon.

To circumvent this, a game with an even smaller state space should be tried. We take the game called Nim, where two players can take one, two or three matches from a pile of matches, but one should not take the last match to win the game. This can be simulated on a 8x8 grid by filling every spot with one match. The state space of the game is 8x8, one state for each number of matches in the game. There is a winning strategy for this game. Always make sure that after your turn ends, there are 4k+1 matches on the playing field.

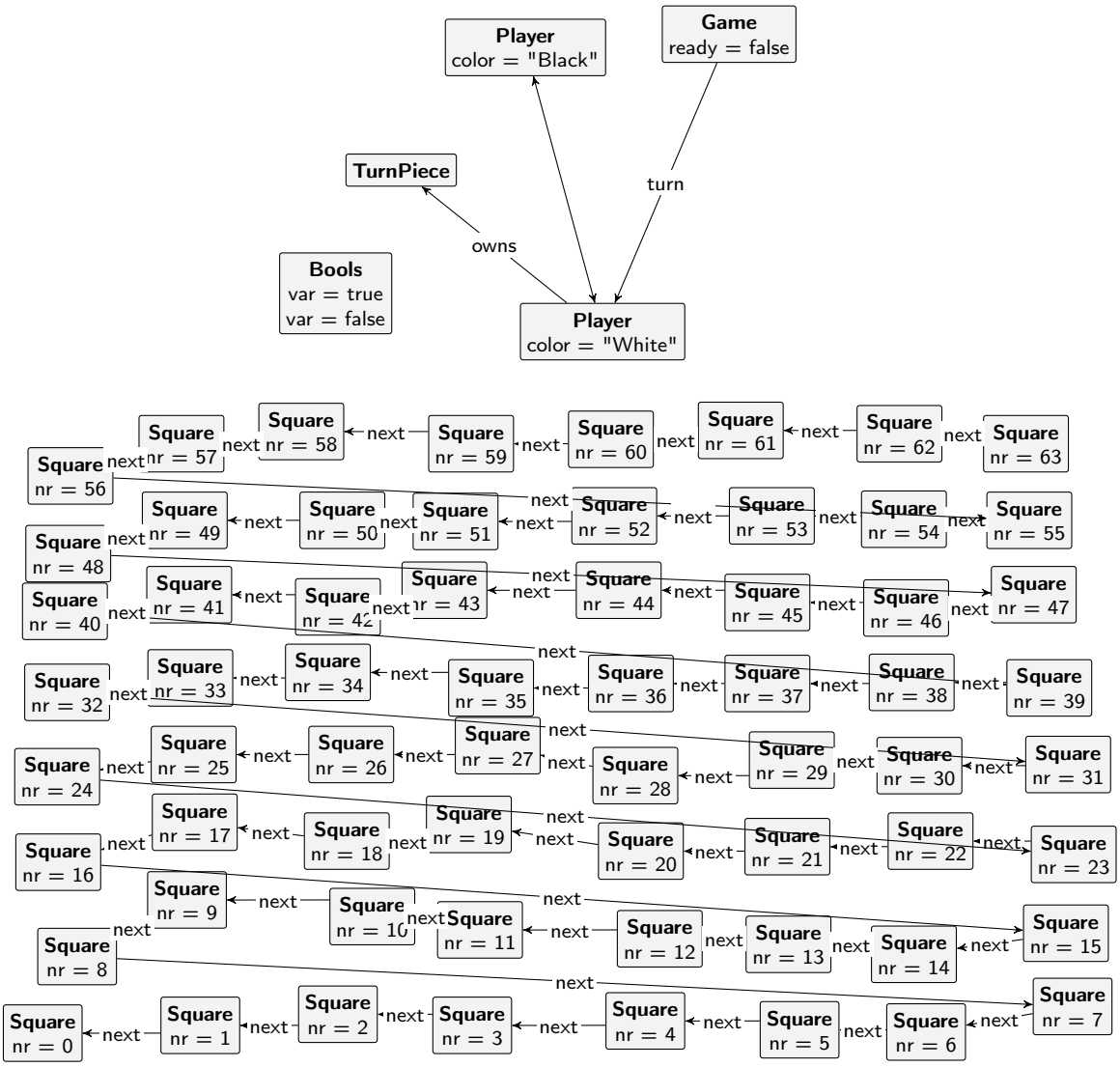

Figure 31: The starting graph for nim.

Figure 31 shows the starting board of nim. It basically is a string of Squares where matches are placed during the 'refreshSquares' transformation. Matches can be removed in order during the game.

Even though with only 64 matches the state space is not too big, Giraffe does not check whether states are the same. Giraffe creates a tree structure for exploring game states, and children are not checked whether they represent the same state. For example, at the start of a game player 1 taking one match, and player 2 then taking two matches, is the same state as player 1 taking two matches and player 2 taking one match from the pile. In GROOVE these states are the same, but in Giraffe they are not. This makes reaching a game ending state a challenge, as the same states and all its children are evaluated multiple

times. Usually, when a game is played against a CPU player, the CPU player will initially not find the winning strategy, as the search tree of game states is not deep enough to find a ending game. But, eventually the game will find these states, and the winning strategy will be found and followed.

Another game that has been implemented to work with Giraffe and GROOVE is Tic Tac Toe. This game too has a small state space. There is no winning strategy for this game, but it's always possible to fend off a loss from the start of a new game. When running Giraffe with GROOVE with this game, this is exactly what happens. The CPU player will not find a win, but will find a way to draw the game, as that is the best way to play the game.

4.1.5 Summary

Performance is still a factor in this use case, which makes evaluating chess moves impossible. For other games that have a smaller state space, the lower performance is manageable. The use case shows that Grammars can be interchanged, and without recompilation of Giraffe, the game it learns and plays can be changed.

4.2 ArmSimulator (CPU assembly simulator)

This subsection the covers use case where ArmSimulator gets coupled with the GROOVE API.

4.2.1 The project

ArmSimulator[23] is a project for educational purposes that simulates a CPU of the ARM instruction set. CPUs are components in a computer that process instructions that do computation on values, which are stored in memory and registers. CPUs are also called processors.

An instruction is a code which does a specific computation on a CPU. The ARM instruction set has the ADD instruction, which takes the contents of two registers, or a register and an arbitrary number, adds them together and puts the result in a third register parameter. Different CPUs have different sets of instructions they support. In PCs, x86 is the most prominent instruction set used at the moment, in mobile phones it is ARM.

In the GUI of ArmSimulator, instructions can be typed, where after they can be simulated. The GUI shows the content of registers and memory during the simulation. In other words, the program can be used to run ARM instructions and check what the effect of those instructions are.

4.2.2 ARM

ARM[1] is an instruction set mainly used in embedded devices, like for example smartphones.

ARM instructions work in processors designed for that instruction set. It is register-based, which means it has an number of storage slots where results of computations can be stored. ARM processors have 16 of these slots, or registers as they are called in processors. All registers are accessible for every instruction.

4.2.3 JVM bytecode

JVM bytecode[7] is another instruction set. This one can be run in the Java Virtual Machine (JVM), which is a software package that can be installed on many different platforms.

JVM bytecode is fundamentally different in design from ARM, because it is a stack-based instruction set instead of a register-based one. Stack-based instruction sets, in contrast to register-based ones, do not have a fixed size of how much it can store. The trade-off is that a stack can only increase in size at the top, and also can be observed and decreased from the top.

4.2.4 Connection with GROOVE

The use case we defined is that we can replace the simulator part with a GROOVE Grammar. GROOVE should be able to simulate processors itself. A starting state is created for a CPU with empty memory and registers, and each supported instruction is modeled as a graph transformation that changes those values. ArmSimulator will use its usual GUI, but all logic for the simulated CPU is transferred to GROOVE.

A drawback for this project is that ArmSimulator needs a parser, which is ARM-specific. It enriches the editor with code highlighting, but most importantly, it converts the instructions from a text form to a data structure a machine would understand. It cannot easily be replaced with the features GROOVE has to offer.

4.2.5 Use case expectations

We consider this use case to be a success if:

1. ArmSimulator uses an ARM GROOVE Grammar, and a connection to GROOVE, for simulating an ARM processor.
2. The GROOVE Grammar can be replaced with one representing the JVM bytecode instruction set and ArmSimulator can work with that.

An important shortcoming will be that the ArmSimulater for JVM will have a different codebase because of the change in parser. This means not only the Grammar is interchanged, but a whole different binary for ArmSimulator is used.

4.2.6 Implementation

ArmSimulator with ARM Grammar

The first step for combining ArmSimulator with GROOVE is removing all ARM logic and replacing this with GROOVE logic. An ARM Grammar was created in GROOVE, so that simple assembly programs are able to run on the new ArmSimulator. The Greatest common divisor algorithm has been implemented for this for the ARM instruction set.

```
main:
mov r1, #6
mov r0, #8
while:
cmp r0, r1
beq end
cmp r1, r0
bgt if
b else
if:
sub r1, r1, r0
b while
else:
sub r0, r0, r1
b while
end:
bx lr
```

Figure 32: Greatest common divisor of 6 and 8, written in ARM assembly.

Some instructions in ARM have a parameter for an arbitrary integer. The instruction set has been implemented in a GROOVE Grammar in two ways.

1. Use extra nodes with possible numbers.
2. Use the MATCH command (Dialog Oracle) to choose a number.

For both solutions we look at the ADD instruction where the content of a register and an arbitrary number (immediate number) are added and stored in

a third register parameter. The two registers are simulated using nodes which contain a register number and a value that is stored in the register. Simulating the immediate value is a problem that can be solved in different ways.

The first solution creates for a node for every possible integer, which is in the range of 0 till 4096 for this instruction. The nodes have its number stored as an integer connected to the node. Now the transformation rules are made to match with a register node, a possible integer node, and another register node, as its parameters This means that the total number of possibilities for this instruction is 16*16*4096=1048576. This is very hard for GROOVE to calculate, and makes the performance of evaluating instructions slow. Moreover, having 4096 extra nodes overhead makes exploring the graph for transformation expensive for every rule. The Grammar can perform when only a few integer nodes are created, lets say 20. This however makes it impossible to do sums with immediate values bigger than 20.

In the second implementation of the Grammar, we use the MATCH command to provide the transformation with an immediate value. So if 'ADD r1 r2 30' is required, the 30 can be provided with the MATCH command and the instruction can be applied. When the processor for some reason would return to the same state as before (for instance by reverting the contents of the registers and memory), 'ADD r1 r2 40' would not be possible, because the 30 from the last ADD is preserved.

In order to make the Dialog Oracle work for the times the simulation returns to an already visited state, we introduce a dedicated counter that increases every time an ask parameter is evaluated. This way the state will always be different.

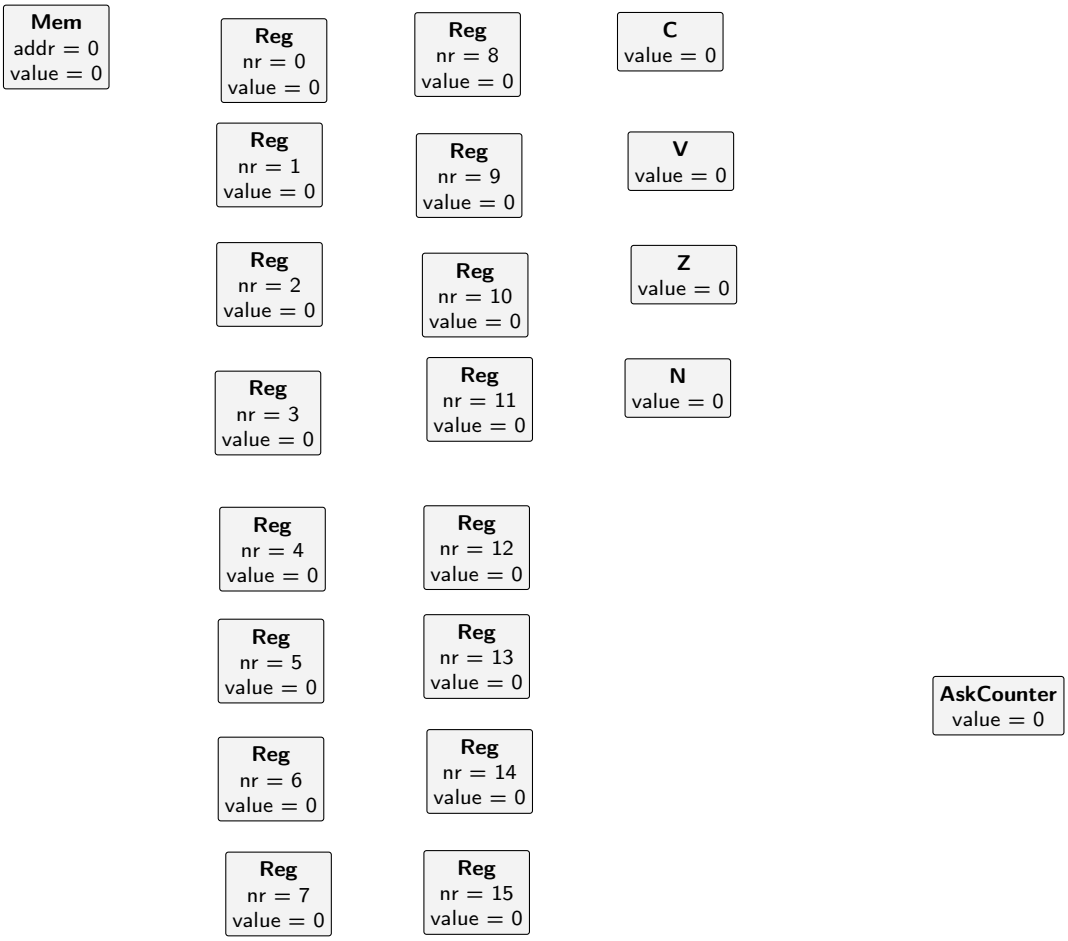

Figure 33: Starting graph for the ARM instruction set.

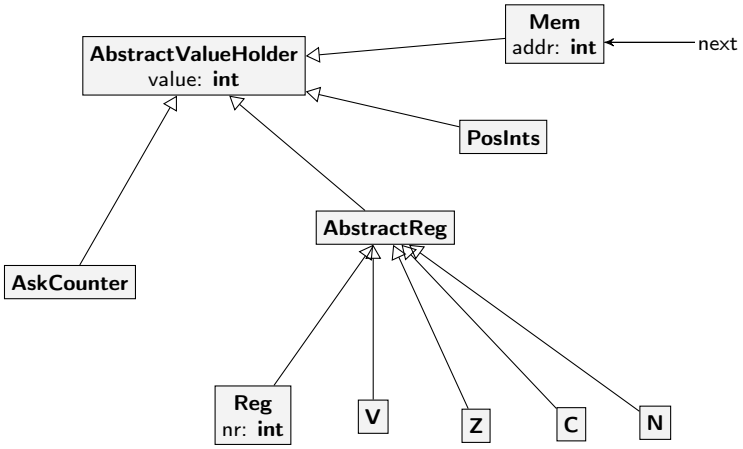

Figure 34: The Type graph for the ARM instruction set.

Figure 33 shows the starting graph for an ARM processor. This is the Grammar which makes use of the Dialog Oracle to implement immediate values. The processor supports 16 general purpose registers and 4 condition registers. 'Mem' contains the top addressable position of the memory. Before a simulation is started, a rule called 'init' is used to expand this memory to be 4096 bytes long. AskCounter is used to increase its value every time a Dialog Oracle is used. Figure 34 shows the Type graph the ARM Grammar uses.

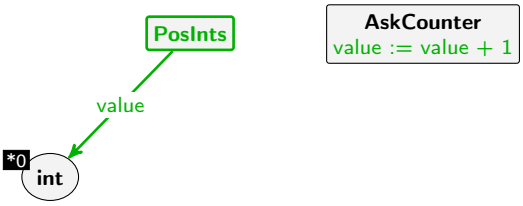

Figure 35: 'ask' rule of the ARM Grammar.

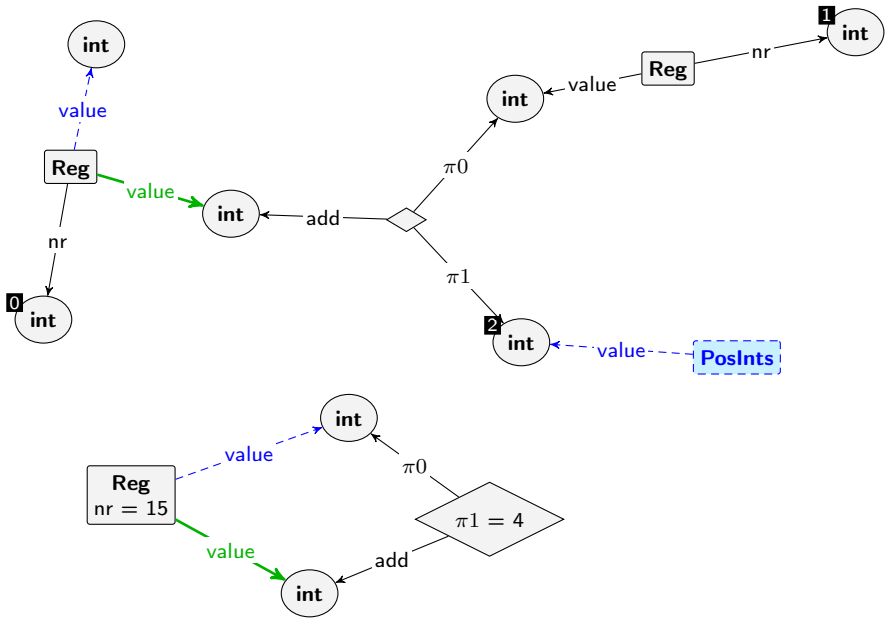

Figure 36: 'add' rule of the ARM Grammar.

Applying a graph transformation to do an 'add' instruction requires two rules. The first is the 'ask' rule (Figure 35), which makes use of the Dialog Oracle to request a immediate value and puts is in a new PosInts node, and coincidentally increases the AskCounter. After this is done, the 'add' rule (Figure36) is used to consume this PosInts value and add this to the requested register. It also increases the the program counter by four (register 15).

ArmSimulator with JVM Grammar

In order to find the true benefit of using GROOVE for offloading the the simulation logic, we need to take a look at replacing the instruction set with another one.

Supporting another instruction set requires writing a new parser for that instruction set, in order for ArmSimulator to recognize the text of instructions. The parser is the only instruction set specific component in the code for ArmSimulator using the GROOVE API. Other instruction-specific code is contained in a GROOVE Grammar. This new parser for a different instruction set and loading another GROOVE Grammar enables ArmSimulator to run simulations for another instruction set. ArmSimulator is very dependent on an instruction set parser to interpret the input text where the user of ArmSimulator can write instructions. Unfortunately this requires changes in the ArmSimulator code, and therefore a recompilation of its code. For this project a GROOVE Grammar which supports JVM is created.

Nevertheless a version of ArmSimulator has been built that uses a Grammar for JVM to perform instructions, but to achieve this a parser for the JVM instruction set was written, and minor changes to the GUI were made. This has the unfortunate effect that changing the loaded grammar for one with a different instruction set is not enough to change the behavior. However, in case for one instruction set an instruction is implemented incorrectly, fixing the bug in the Grammar and reloading it will solve the problem without recompiling ArmSimulator.

```
main:
bipush 6
istore 00
bipush 8
istore 01
start:
iload 00
iload 01
if_icmpeq end
iload 00
iload 01
if_icmpgt else
iload 01
iload 00
isub
istore 00
goto start
else:
iload 00
iload 01
isub
istore 01
goto start
end:
iload 01
ireturn
```

Figure 37: Greatest common divisor of 6 and 8, written in JVM bytecode.

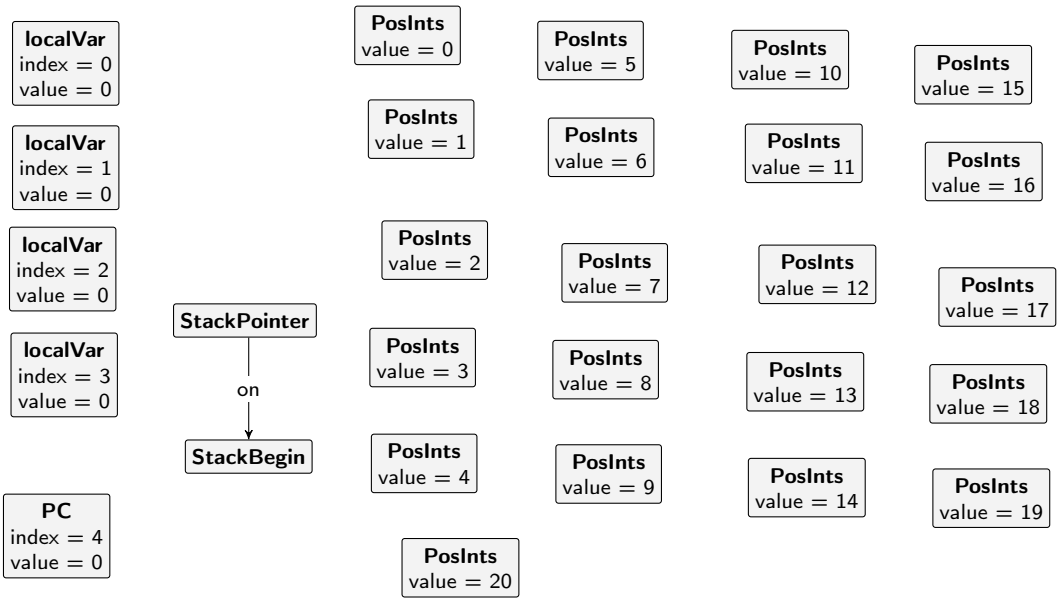

Figure 38: Starting graph for the JVM instruction set.

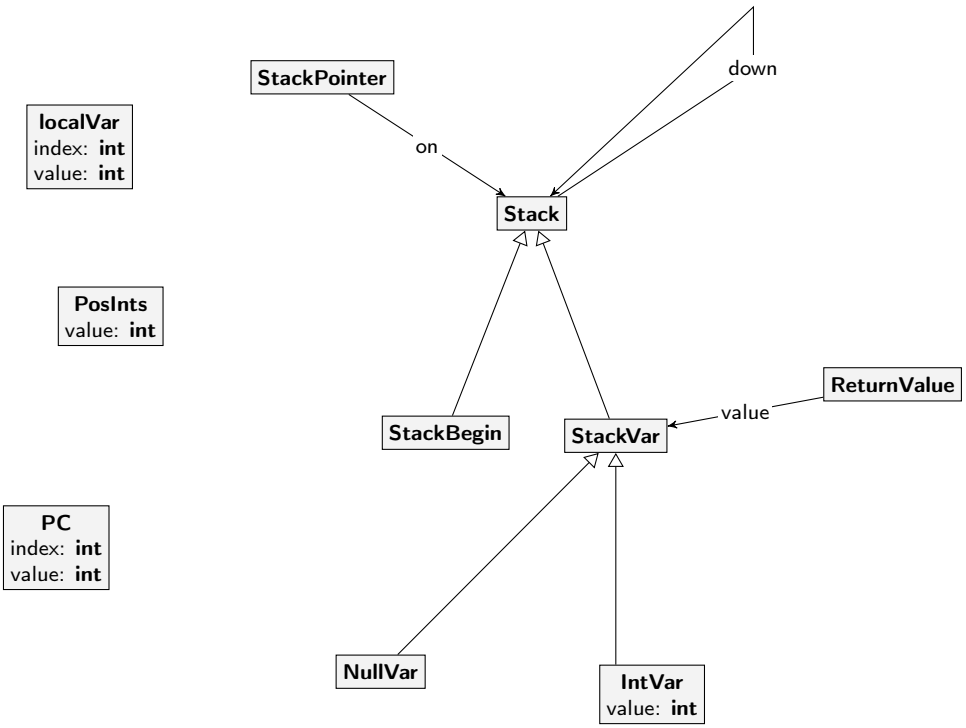

Figure 39: The Type graph for the JVM instruction set.

Figure 38 shows the starting graph for the JVM. This is the Grammar which

makes use of the nodes with the possible integer values to implement immediate values. JVM uses 4 general purpose register (localVar) and a stack to store values. Figure 39 shows all types used in the implementation of this Grammar.

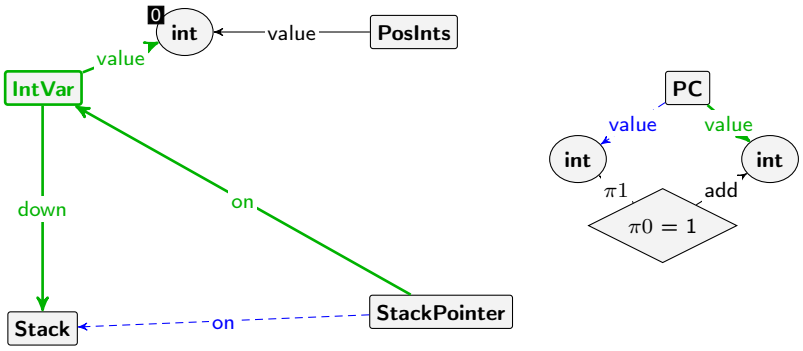

Figure 40: The 'bipush' instruction of the JVM instruction set.

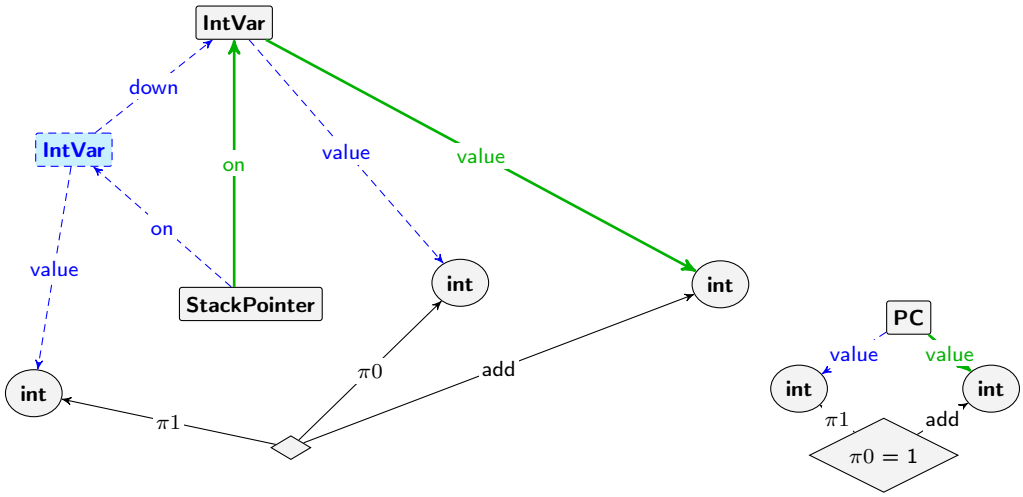

Figure 41: The 'add' instruction of the JVM instruction set.

Adding two number on the JVM takes two different instruction. Numbers can be stored on the stack with the 'bipush' instruction (Figure 40). It creates a new IntVar to store the new value and moves the StackPointer to the top of the stack. It also increases the program counter. When the top two values on the stack are of type IntVar, the 'add' instruction can be performed (Figure 41. It takes the two numbers, adds them together and puts that back on the stack. It also adds one to the program counter.

4.2.7 Summary

In this case is shown that GROOVE can work in combination with ArmSimulator, in both the situation where the ARM is simulated in a Grammar, and the situation where JVM is simulated, although both instruction sets run on different binaries of ArmSimulator.

For the ARM version of ArmSimulator the two Grammars can be used. One which uses integer nodes for selecting immediate values, and one which uses the Dialog Oracle to select immediate values. The first one has the problem of having a small set of possible integers. With this shortcoming, the performance is way worse than the original C++ version. Small programs take seconds to finish, where the original code could do this in an instant. The second Grammar can at any moment use any immediate value, but still, simple programs take multiple seconds to finish. This part of the use case shows that the ARM logic can be replaced with a GROOVE Grammar. Moreover, two versions of the Grammar are created which are inter-changeable without any recompilation.

The JVM version has a single Grammar implemented. It shows that existing instruction logic can be replaced with a GROOVE Grammar. It also has the flaw that the speed of performing instructions is not that high.

This use case shows that parts of the business logic can be deferred to the GROOVE API, however the speed is worse compared with the original project.

The main advantage of using the GROOVE API is that bugs in instructions can be solved in the grammar, and do not require any recompilation. A side effect of using GROOVE is that the speed of performing instructions is lower then that of the original C++ code.

The main conclusions of this use case are:

1. The performance is not good enough for assembly execution
2. The changes in user interface are not easily captured in a GROOVE Grammar

4.3 Lego Mindstorms EV3

This subsection covers the use case where the hardware of Lego Mindstorms is used in combination with the GROOVE API.

4.3.1 The project

Lego Mindstorms EV3 is a modular hardware structure. The hardware consists of components which can be combined to produce a driving robot. The components include sensors, like a touch sensor, which can detect whether the

robot is touching a wall or something similar, or a gyrosensor, which detects the orientation. Additionally, there are actuator components, which make the hardware able to interact with the environment, like motors, or linear actuators, which can be used to build all kinds of contraptions.

4.3.2 Connection with GROOVE

The default firmware of the EV3 is limited in its capabilities. It is configured using the provided proprietary LEGO MINDSTORMS Education EV3 software, and is not customizable. We need to be able to run custom code to respond to specific instructions from the PC it is connected to. We need that connection to run GROOVE on a external system, because the EV3 cannot run it itself.

In the past the Formal Methods and Tools department of the University of Twente has done research on testing programs running on an NXT, the precursor of the EV3[14]. In this research the NXT is running leJOS[5], which is a firmware that replaces the default firmware on the hardware, and enables a programmer to write Java programs that run on the hardware. It is also possible to use a USB connection to an external computer. This software can also run on the newer EV3. We write a program for leJOS that can connect to a program on the PC that has a GROOVE backend. The EV3 program is able to gather all information it knows from its sensors and sends it over this USB connection. Additionally, it is able to receive messages which contain the instructions for the actuators. The PC program with the GROOVE backend will function as a message bridge between GROOVE and the EV3's USB connection.

We developed a program with a GROOVE backend that runs on the external computer. Using the GUI of GROOVE, a GROOVE Grammar can be constructed which can be used in the use case. The Grammar will contain information about actuators and sensors that are on the EV3.

So, two custom programs are needed for the setup. The first custom program runs on the external computer, and will load the Grammar and connect to the EV3. The first custom program functions as a bridge between GROOVE and the program running on the EV3 hardware. This bridge makes sure the Grammar is loaded, and is responsible for the process of receiving sensor data and determining new actions for the actuators.

The second program is running on the EV3. The EV3 sends its sensor data and the GROOVE Grammar determines, based on that data, what the next state of the actuators should be. The EV3 then receives new actuator actions, which it will perform.

4.3.3 Use case expectations

This use case is used to show that GROOVE is capable of describing sensors and actuators of an EV3, and that loading different grammar models will result in different behavior of the hardware. No programming is needed to get different

behavior, only loading a new grammar. For this use case, basic grammars will be constructed that simulates a few sensors and actuators, and defines a basic relations between the components.

We consider this use case to be a success if:

1. The Lego Mindstorms hardware can be controlled by a GROOVE Grammar using a GROOVE connection
2. Different behavior can be given to a LEGO setup, by interchanging the underlying GROOVE Grammar

4.3.4 Implementation

For the grammar construction, the graph contains every sensor and actuator as a node. Additional information like port numbers and sensor/actuator specific values like motor speed are properties of those nodes.

There is no detection of the sensors and actuators that are connected to the EV3, which means that every GROOVE Grammar needs its attachments and port numbers changed every time something changes. This is the same case when programming the EV3 without GROOVE. It is a shortcoming of the EV3 hardware.

The program that is running on the EV3 (EV3Groove) waits for a connection to be established by a PC. If a connection is made, it waits for it to be instructed which sensors and actuators are connected to it. After that it will wait for instructions for actions for the actuators. Simultaneously, it will keep sending the current detected values from the sensors to the PC.

The second program (EV3GrooveBridge) is a bridge between the EV3 and Groove. The program will begin with connecting to GROOVE and loading a predetermined Grammar. After that is done, this program will connect to the EV3. The Grammar contains the sensors and actuators that should be connected to the EV3. The bridge program will send this information to the EV3. When this setup is done, it will start receiving the sensor data from the EV3, and using the 'set' rule to change the Graph in GROOVE to set the sensor data. Next, the 'next' rule updates the actuator properties based on the rule implementation and the current graph state. These actuator properties will be extracted from the new state of the graph, and sent to the EV3.

The LEGO setup tested consists of one color sensor and one motor actuator. The color sensor determines the color of a ball, after which the motor with spokes attached sorts the ball to the left or right, based on the color. The grammar can be changed to support different colors, or to switch what goes right and left.

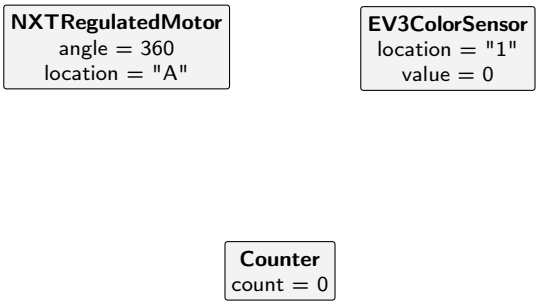

Figure 42: The starting graph for the LEGO Mindstorms setup.

Figure 42 shows the starting graph for the LEGO Mindstorms setup. This one only consists of three nodes, which are the two components that are part of the LEGO setup (NXTRegulatedMotor and EV3ColorSensor), and a counter to always create an undiscovered state whenever the Dialog Oracle is used. The two components have a location value. This is the port where the component is connected to on the EV3 brick. 'value' is where the sensor data is stored. EV3GrooveBridge makes sure this is set to the current observation. 'angle' is the angle should be. This is what EV3GrooveBridge extracts and sends to the EV3 brick.

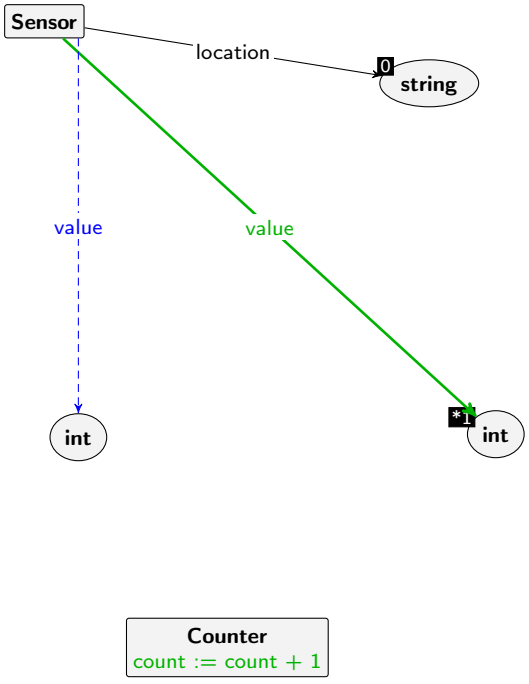

Figure 43: The 'set' rule for the LEGO Mindstorms setup.

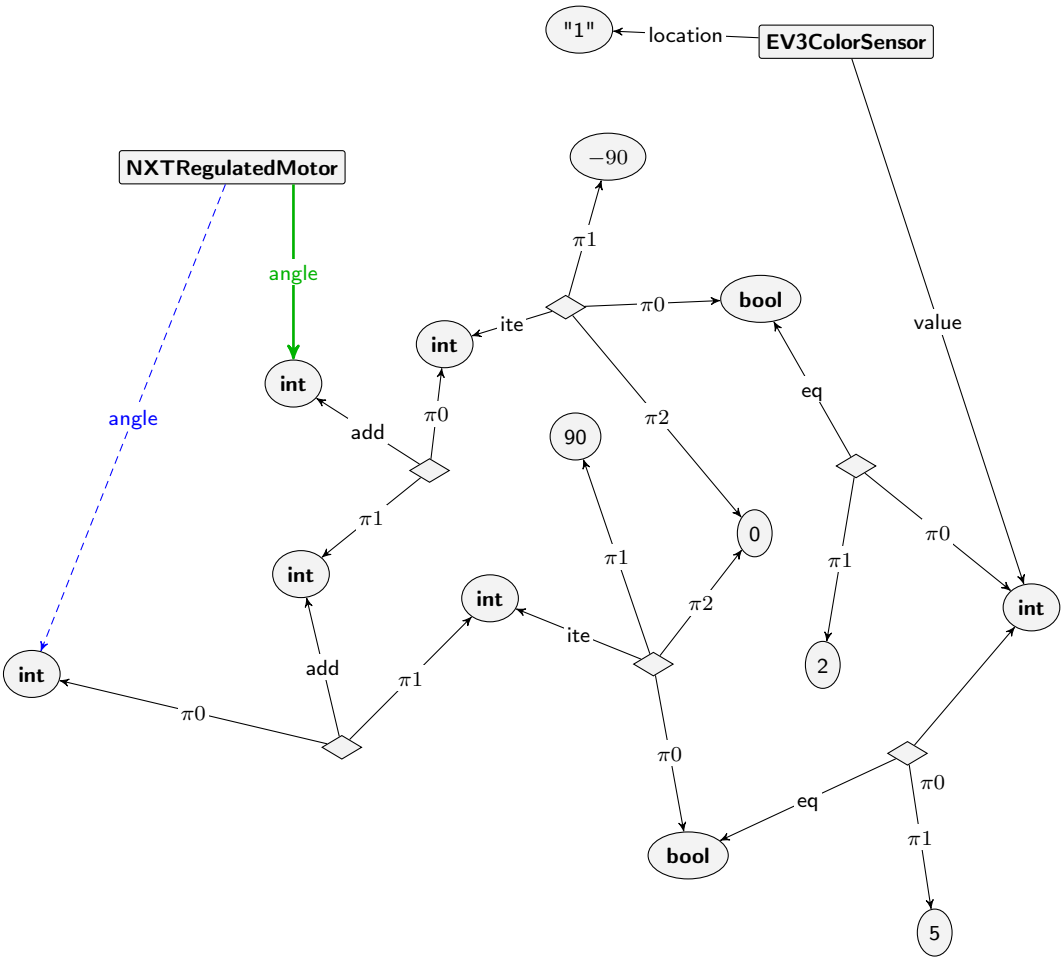

Figure 44: The ‘next’ rule for the LEGO Mindstorms setup.

The Grammar has two rules, ‘set’ and ‘next’. The set rule is used by EV3GrooveBridge to communicate the observations of the sensors to the current state of the graph simulation. The ‘set’ rule (Figure 43) uses the Dialog Oracle to replace the current value with a new value. This rule should not change for different setups. The ‘next’ rule (Figure 44) will be different for different setup. This contains the business logic for what the creator wants to achieve with the Grammar. In this example, the observation of the color sensor is checked, and if it is 2, which is blue, or 5, which is red, it will move the motor 90 degrees clockwise or anticlockwise, thereby sorting the balls by color. The implementation of this ‘next’ rule can be changed to support different colors, but it is also possible to attach different sensors and actuators, as long as they are present in the starting graph. EV3GrooveBridge will detect based on the starting graph what components it should expect.

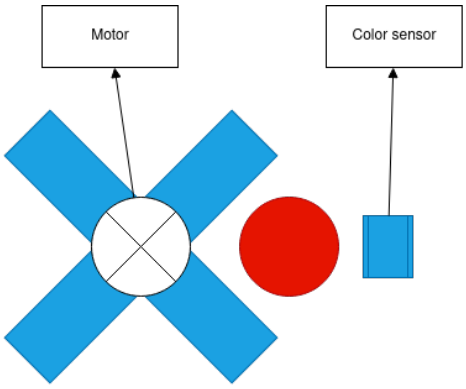

Figure 45: The top view for the ball sorter.

For the ball sorter, the setup looks as in Figure 45. From this top view the ball is dropped where it is in the diagram. There the color sensor will detect it, and act as the 'next' rule. In this case, move the motor by 90 degrees, and move the ball with the extensions that are attached to the motor.

4.3.5 Summary

This use case shows that GROOVE is able to dynamically simulate different LEGO constructions. The GROOVE Grammars can contain enough information for describing simple scenarios.

The main conclusion of this use case are:

1. Performance is not really essential in this use case, so GROOVE is fast enough
2. Different grammars can run different LEGO setups without recompiling any code.

5 Conclusion

This thesis describes how the program GROOVE, which was mainly interactable using a GUI, gained an API, and how this was verified by connecting the API to existing programs. During this progress conclusions have been drawn.

One of the conclusion it that the performance of GROOVE, but probably graph rewriting over all, is not high enough to solve general logical problems. The process of evaluating rules does not use optimal algorithms. Rules in GROOVE very easily get algebraic complexity. The original programs of the use cases implement a better complexity. This was the case for the use cases Giraffe and ArmSimulator.

The way of designing Grammars for general problems is intuitive for most programmable logic. However, the implementation of the ask parameter is reasonable in the context of a state machine, where every transitions is predictable. In the context of general logical problems is has shown to be an obstacle. This was the case for the use cases ArmSimulator and LEGO MindStorms.

The final conclusion is that GUIs, and TUIs for that matter, are not easily automatically altered when a GROOVE Grammar is interchanged in an application that is using the GROOVE API. The details of parts of the GUI could be contained in the Grammar, so the program can construct the GUI with those details. However, this structure was not used in these use cases because of the increase in complexity it gives. This was the case for the use case ArmSimulator.

The goal was to implement an API for GROOVE, and requirements (section 3.1.1) were set for implementing this API. The requirements suffice the functionality for the tested use cases in making use of GROOVE, and extrapolating from that, similar programs that want to use GROOVE will too.

5.1 Reflection

This thesis outlines how we extracted a subset of the components of GROOVE and package it as an API for other programs to use. Besides that we showed use cases which validate and reflect the usage of the API against.

The deliverables of the project are:

1. A program in the GROOVE project, which provides an API. This is written in Java.
2. A change in the Giraffe program, which enables it to use GROOVE as a backend. Giraffe is written in C++.
3. GROOVE Grammars for chess, connect four, nim and tic-tac-toe.
4. A change in the ArmSimulator program which enables it to use GROOVE as a backend. ArmSimulator is written in C++.

5. GROOVE Grammars for the ARM instruction set and the JVM bytecode instruction set.

6. A new program that runs on the Lego Mindstorms hardware (EV3Groove), which will connect to a PC via USB. This is written in Java.

7. A new program that runs on a PC (EV3GrooveBridge), bridging a connection to GROOVE and a connection to Lego Mindstorms hardware. Written in Java.

8. GROOVE Grammars for deciding Lego Mindstorms movements.

5.2 Future work

One of the shortcomings of the design of GROOVE when used for the applications in the previous use cases is the implementation of the 'ask' parameter. The logic for the current design is defensible, but the design does not fit in a environment where GROOVE rules can be called with a arbitrary integer for example. For simulating where parameters can be arbitrary, this design does not fit.

About the choices made for the API: the protocol is pretty custom right now, and for showing the purposes and possible interactions of a stateful GROOVE API this did the job. If this API was to become more widely used, using a framework to create a stateful connection can be beneficial for developers of client applications in order to connect to the GROOVE server. This should add more boilerplating to the implementation of the API and decrease the chances on programming mistakes.

If the Giraffe use case would be explored further, it would be an addition to create a way to add game specifics to the neural network feature set, by adding those specifics in the Grammars of those games and implement a way in Giraffe to extract those specifics.

The GROOVE API is mainly designed to be useful for simulating Grammars, but it cannot create new Grammars. The API could be extended, to also be able to create Grammars.

References

[1] Arm architecture reference manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest.

[2] Extensible markup language (xml) 1.1 (second edition). https://www.w3.org/TR/xml11/.

[3] Java native interface (jni). https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.

[4] Java remote method invocation (java rmi). https://www.oracle.com/java/technologies/javase/remote-method-invocation-home.html.

[5] lejos. http://www.lejos.org/.

[6] GROOVE. https://groove.ewi.utwente.nl/.

[7] The Java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se8/html/.

[8] The LAME project. https://lame.sourceforge.io/.

[9] Standard: Portable game notation specification and implementation guide. https://ia902908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt.

[10] File Transfer Protocol - meeting announcement and a new proposed document. RFC 454, Feb. 1973.

[11] User Datagram Protocol. RFC 768, Aug. 1980.

[12] Transmission Control Protocol. RFC 793, Sept. 1981.

[13] Telnet Protocol Specification. RFC 854, May 1983.

[14] A. Belinfante. Automatic (model-based) testing of a lego ball sorter. https://fmt.ewi.utwente.nl/lego-sorter/.

[15] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, Dec. 2017.

[16] G. K. A. L. N. M. H. F. N. S. T. D. W. Don Box, David Ehnebuske. Simple object access protocol (soap) 1.1. Simple Object Access Protocol (SOAP) 1.1, May 2000.

[17] R. T. Fielding. Architectural styles and the design of network-based software architectures, 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[18] R. T. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.

[19] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[20] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648, Oct. 2006.

[21] M. Lai. Giraffe: Using deep reinforcement learning to play chess. September 2015.

[22] C. M. Lonvick and T. Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, Jan. 2006.

[23] T. Schulz. ArmSimulator. `https://github.com/MrDiver/ArmSimulator`.

[24] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Aug. 1995.