Bachelor Thesis

Visualizing BTOR2 Models

Markus Diller

Advisor: Christoph Kirsch

Department of Computer Science
Paris Lodron University
Salzburg, Austria

August 2022

Abstract

The process of bounded model checking is very helpful in finding edge cases for invalid program inputs. To be able to do this, a format representing the model is necessary. Unfortunately, the model file itself does not provide any information about metrics or quality of a model. Its only purpose is to serve as an input to a model checker. In this thesis a tool called `beatle` for a subset of the BTOR2 modeling language is described. It helps to alleviate the aforementioned limitations of a standalone model file by visually displaying relevant information in the browser.

Contents

1 Introduction 2

2 Prerequisites 3

2.1 Selfie system 3
2.2 Bounded Model Checking (BMC) 5
2.3 BTOR2 5
2.4 BEATOR2 5
2.5 Modeling RISC-U Binaries 7

3 Program 10

3.1 Environment 10
3.1.1 Used Libraries 12
3.2 Parsing and Processing 13
3.2.1 Local Calculations 14
3.2.2 Global Calculations 15
3.2.3 Displaying the Calculations 15
3.3 Visualization 16
3.3.1 Displaying a Graph 16
3.3.2 Abstracting Complexity 19
3.3.3 Highlighting the Longest Paths 22
3.4 Putting it all together 23
3.5 Known Issues 27

4 Conclusion and Outlook 28

5 References 29

1 Introduction

When creating software, one important aspect is the one of testing a program for different inputs. This is to catch unintended behavior which could lead to unforeseen consequences when deploying the code. One of the most difficult tasks when creating tests is to find meaningful inputs that cover all possibilities the program could encounter. Often times, this is a cumbersome and arbitrary task when performed by humans. Therefore, it would be advantageous to use a more bulletproof approach to find these inputs.

Bounded Model Checking is one such approach which helps find so-called witnesses, that create unwanted outcomes for a program. A program cannot be put into a model checker as is, but needs to be converted into a proper format first. The format at the center of this thesis is a subset of BTOR2[6], from now on called BEATOR2, generated by the `BEATOR` tool. Essentially, it is just a text file containing the information for a bounded model checker to fulfil its purpose. Alas, a user generating a model from a program binary has no idea what the model even contains, due to the convoluted appearance of the file's contents.

It may not even be the case that the model actually is equivalent to the input binary. There could possibly be bugs in the conversion which are almost impossible to debug without a great deal of patience when just inspecting a wall of text and numbers. A less extreme case may be that the model does what it should but not in a very efficient manner. Finding these problems with just the file is not feasible in a reasonable amount of time. As a ballpark figure, even the most simple binary converted through the `BEATOR` tool contains at least 600 lines of model information when not applying any optimizations.

The `beatle` tool described tries to support users creating models and developers programming the conversion software to grasp the makeup of generated model files. It achieves this by visually displaying the model as a directed acyclic graph, and provides information about every node, as well as the entirety of the model. The tool also introduces abstraction for different parts of a model and helps with following flow throughout a graph.

With this tool there is also the possibility of better understanding the underlying model format itself. The model itself is not presented as an unwieldy collection of characters in a terminal window, but as an interactive and customizable webpage in the browser, easily accessible without any installations. The interface follows an easy to pick up structure with the relevant information being properly labeled. This means that the tool can be used as a companion when teaching and learning the BEATOR2 model format. For that purpose there is an example model provided, which can be effortlessly loaded.

2 Prerequisites

Before diving into the nitty-gritty of describing the tool at the center of this thesis, some background information is necessary. It is, however, only the relevant information for understanding the following chapters and is therefore not an exhaustive but a rather shallow introduction into the presented topics.

2.1 Selfie system

The `BEATOR` tool, which serves as the foundation for creating BEATOR2 models does not accept any binary file. It is restricted to the RISC-U instruction set. As the name already suggests it is a derivate of the popular RISC-V instruction set, to be more specific, it is a subset. In order to generate binaries in the RISC-U format another tool is necessary, called `selfie`. One of its many purposes is to compile C* code into valid RISC-U binaries.

C* is yet another subset of a language, this time the programming language C. It contains five statements, comparison and arithmetic operators, and features only two data types, one representing 64 bit integers and the other 64 bit pointers. The selfie system supports a few system calls, the most important ones being `brk()` or `malloc()` for dynamic memory allocation, `read()` for reading external information and `exit()` for terminating a program.

One of the simplest C* programs, which is also the one used as the converted program for the example model in the `beatle` tool, can be seen in Figure 1. In it there exists a global variable *x*, a local variable *a* and one instance of the aforementioned system calls, as well as three assignments. The exit system call is implicitly called at the end of the `main()` procedure. The program only successfully terminates with an exit code of zero and without throwing a segmentation fault when '0' is read into memory at the address of *x*.

```
uint64_t* x;        // global variable

uint64_t main() {
  uint64_t a;       // local variable
  x = malloc(1);    // dynamic memory allocation
  *x = 0;           // initialize data at address of x
  read(0, x, 1);    // reading input from console into x
  a = *x - '0';     // map ASCII number to actual number

  return *(x + a);  // try to access memory at address x + a
}
```

Figure 1: Simple C* program.

The RISC-U binary of this program code can be generated by the `selfie` tool in the command line by invoking `./selfie -c example.c -o example.o`. The instruction set contains 14 instructions and the machine contains a program counter, as well as 32 different 64 bit registers. System calls are invoked by loading the necessary parameters into registers `a0` through `a3` and the system call ID in register `a7` followed by an `ecall` instruction. An overview of the instructions in addition to their semantics can be seen in Table 1. The generated out file is then fed into the `BEATOR` tool to generate the final BEATOR2 model file.

Instruction	**Semantics**
`lui rd,imm`	$rd = imm \cdot 2^{12}$
`addi rd,rs1,imm`	$rd = rs1 + imm$
`ld rd, imm(rs1)`	$rd = memory[rs1 + imm]$
`sd rs2, imm(rs1)`	$memory[rs1 + imm] = rs2$
`add rd,rs1,rs2`	$rd = rs1 + rs2$
`sub rd,rs1,rs2`	$rd = rs1 - rs2$
`mul rd,rs1,rs2`	$rd = rs1 \cdot rs2$
`divu rd,rs1,rs2`	$rd = rs1/rs2$
`sltu rd,rs1,rs2`	$rd = \begin{cases} 1 & if rs1 < rs2 \\ 0 & else \end{cases}$
`beq rd,rs1,rs2`	$pc = \begin{cases} pc + imm & if\ rs1 = rs2 \\ pc + 4 & else \end{cases}$
`jal rd,imm`	$rd = pc + 4$
	$pc = pc + imm$
`jalr rd,imm(rs1)`	$rd = pc + 4$
	$pc = \frac{rs1+imm}{2} \cdot 2$
`ecall`	performs system call

Table 1: Overview of the 14 instructions. `rd` is the destination register, `rs1` and `rs2` the source reigsters, `imm` an immediate value. If not specified, the PC is incremented by four

All tools and language formats described in this section are part of the selfie ecosystem[4]. They are part of the initiative for educating students in the University of Salzburg on the topics of system engineering and are developed and maintained by the Computational Systems Group at the Department of Computer Science[5].

2.2 Bounded Model Checking (BMC)

The BEATOR2 format is applied when conducting Bounded Model Checking. Model Checking describes a technique used for verifying finite state systems, e.g. sequential circuits. A model is specified in temporal propositional logic and the changing values are modeled as a state transition graph. The big advantage of using this verification approach is the highly automatic nature. A user evaluating a system only needs to provide the model and a specification to check for. The model checker then either finds a counterexample that contradicts the specification or that the model meets the specification [2].

The predecessor to Bounded Model Checking was Symbolic Model Checking which uses binary decision diagrams that do not scale well for more than a hundred variables. Therefore, BMC tries to leverage the increased reasoning prowess that SAT formulae provide. Unfortunately, BMC loses the ability to prove the absence of a counterexample. This is due restricting the amount of executions to a specified bound [1].

When applying this definition of Bounded Model Checking to the specific use case for the BEATOR2 format, modeling RISC-U binaries, it can be defined as follows. We want to model arbitrary RISC-U binaries as sequential circuits with certain safety properties, like zero as an exit code. A bounded model checker can then verify that for a specific execution depth of the model there is no input that causes the defined properties to be negated, or it finds an example which proves that there is.

2.3 BTOR2

Like the other language specifications in the selfie system, BEATOR2 is a subset of an already established modeling format called BTOR2. It can describe models for software or hardware. What differentiates this format from others is the availability of sequential operations, as well as being able to define data types via sorts [6].

There are three different properties or constraints the format is able to model. These are negated safety properties, fairness constraints, and liveness properties. The specification also includes a definition of the witness format, i.e. the form in which a counterexample for the modeled properties are presented [6].

2.4 BEATOR2

Unlike BTOR2, in the BEATOR2 model format only models for safety properties. These are currently non-zero exit codes, invalid access to memory, i.e. segmentation faults, and division by zero. The format operates on word level and is bit-precise [3].

The model itself is presented as a text file. Every line represents a single operation of the BTOR2 instruction set and consists of the following parts: a number that uniquely identifies the instruction, a keyword to indicate the type of instruction then an identifying number for what data type this instruction operates on and further parameters necessary for the operation which are either more IDs, an immediate or string [5].

```
...
10000574 ite 2 10000162 10000441 10000573
10000575 ite 2 10000557 10000571 10000574
10000576 next 2 10000137 10000575
10000577 constd 2 8
10000578 add 2 10000040 10000577
10000579 constd 2 8
10000580 add 2 10000040 10000579
10000581 state 2 fp
10000582 init 2 10000581 10000029
10000583 constd 2 18446744073709551592
10000584 add 2 10000040 10000583
...
```

Figure 2: Snippet of the generated model for the program in Figure 1.

When performing model checking, colloquially speaking, every line in the file from top to bottom evaluates to a fixed number until a clock tick occurs and the process begins again from the top. As a consequence a value of a line never changes in the same clock tick, meaning this fact needs to be properly accounted for during the modeling phase. The only instruction that is an exception to that rule is the initialization of `state` instructions. These directly modify values and are not evaluated after the first clock cycle.

The initialization instruction seems like a great misfit to the other instruction set and severely hinders the otherwise easiness of parsing this format. Another sub-optimal design decision of this format is that there are two options for allowing variable input to the program. One being the aptly named `input` instruction, the other being uninitialized `state` instructions.

After the clock ticks, the values in the `state` instructions change to the evaluation in the corresponding `next` instructions. This is the sequential part of the format. The other operations then use these updated values in `state` instructions and possibly change their evaluation.

If at any point before the bound any `bad` instruction evaluate to true, model checking can conclude because a witness for a negated safety property, has been found. On the other hand, if no such input has been found until the bound is exceeded, it is guaranteed that there is no input causing any properties to be violated until this bound.

2.5 Modeling RISC-U Binaries

When modeling a RISC-U binary, BEATOR pertains to a certain structure in a model, at least before optimizing the final generated file. For example, every state is tagged with a comment containing a human-readable description which also persists when emitting the final result.

Firstly, the data types in the form of sorts is defined. These contain the definitions for a machine word, boolean, virtual memory and the other necessary data types. Next, the common constants like true or false for boolean, or the 0 as a machine word are defined. Then, the registers, program counter flags and virtual memory as `state` instructions, followed by modeling of the system calls. Second to last, the state transitions are modeled for the control and data flow and lastly the logic for different safety properties.

The exhaustive list of what needs to be modeled for recreating a semantically identical model to the input binary are registers, program counter states, the virtual memory, system calls, and obviously the instructions in the binary itself. At the time of writing this thesis, every possible value for the PC is represented by its own boolean state. This means that at one time only one of these states can be set to true and when transitioning to another state, in which the program counter changes the true PC gets set to false and another set to true.

One important aspect is the modeling of system calls. For a model to be non-deterministic, meaning the outcome of the model checking is not trivial, some kind of input is necessary. When modelling RISC-U binaries this is the `read()` system call. Due to a machine word being eight bytes, one through eight characters can be stored in one word at a time. Therefore, these different options are modeled using sorts for the different number of bytes, input instructions with the corresponding sort and extension of the read value to always be the size of a machine word.

To give an idea how the system call is actually modeled, Figure 3 shows a snippet of the model file for the program in Figure 1 that contains the parts necessary to determine the value read during the system call.

```
...
10000777 ite 2 10000776 10000774 10000773
...
10000902 constd 2 8
10000903 eq 1 10000777 10000902
10000904 input 2 8-byte-input
...
10000935 ite 2 10000910 10000912 10000934
10000936 ite 2 10000906 10000908 10000935
10000937 ite 2 10000903 10000904 10000936
10000938 write 3 10000569 10000901 10000937
...
```

Figure 3: Snippet of read system call modeling. Some details omitted for brevity.

The verbalized description is denoted in the following list:

- If the number of bytes to be read is less than eight set it to that value else be that value, else set to eight (10000777).
- Decide whether the number of bytes to read is eight (10000902-10000903) and read input for 8-byte input no matter what (10000904).
- Determine the number of bytes read in 10000777 and use this information to decide which read input value to use (10000935-10000937).
- Write into the virtual memory (10000569) at the given address (10000901) the given value (10000937)

As can be seen in Figure 3, this process of trying to make sense of the model by inspecting a wall of text and numbers is cumbersome and leads to too much scrolling and searching for numbers.

The only other system call currently modelled with BEATOR is the `brk()` call. In the selfie system the allocation of data on the heap is realized using bump-pointer allocation. To keep track of what value it is, the first word in the static data segment of the virtual memory is reserved for this purpose. This results in the only thing being denoted in the model is the increase of this value defined by the provided amount using state like the registers, but unlike other access to the virtual memory.

For accessing memory in the heap and data segment in the virtual memory, the `read` and `write` instructions are used. The memory itself is represented using a sort which is constructed of an array of machine words indexed by machine words. The two aforementioned instructions allow loading and storing values in the modeled virtual memory, and is changed at most once before a state transition transpires due to the constraints imposed by only one instruction being active at once.

These constraints are being checked by long if-then-else cascades determining what instruction is currently active and with it what value the next instruction is set to.

The rest of the modeling happens in a pretty straight forward way. Each register has its own state which is initialized using the `init` instruction. Initialization is realized using constants, and arithmetic and comparison instructions have a direct model instruction counterpart. Possible constants loaded via `lui` and `addi` in the binary use `constd` in the model instead. In Table 2 all currently supported instructions are denoted, together with which part of the binary it helps to model.

Instruction	**Usage**
`sort bitvec` *size*	data types for states
`sort array` *size*	virtual memory state
`state` *sort name*	registers, PCs, virtual memory and other states
`init` *sort state value*	initialization of states
`next` *sort name*	sequential transition for values of states
`bad` *sort name*	desired properties to check for (e.g. non-zero exit code)
`input` *sort*	variable input
`constd` *sort integer*	constants
`uext` *sort input integer*	extends input to be of word size
`read` *sort memory index*	read from virtual memory
`write` *sort memory index*	write to virtual memory
`add` *sort x y*	direct mapping from binary instructions
`sub` *sort x y*	
`mul` *sort x y*	
`udiv` *sort x y*	
`urem` *sort x y*	
`ult` *sort x y*	
`ite` *sort bool x y*	if-else cascades of PCs and register state transitions
`and` *sort bool bool*	control flow for branching
`not` *sort bool bool*	
`eq` *sort x y*	

Table 2: Available BEATOR2 instructions [5]

Another feature the BEATOR tool, which generates the model file, provides is the option of unrolling. During the model generation a number of clock ticks can be defined which are then simulated before the final generation of the file.

At least at a logical level this means copying the model the amount of times it is executed, greatly increasing the complexity of the initialization of the states.

The resulting trade-off is larger model size to being able to skip the defined amount of ticks when actually performing the checking.

Copying allows for great optimization in some aspects. Unnecessary states are removed when it is deemed they are no longer useful for trying to find witnesses for negated safety properties, i.e. they only depend on constants. Also, only necessary operations for initialization steps are retained, the other copied instructions are discarded. If there would be a variable input necessary when unrolling it is represented using the other way of having input for a model, uninitialized state variables.

The unrolling procedure naturally results in a larger model file, but not as large as just naively copying the model the amount of times specified. Displaying the effectiveness of this unrolling technique was also a goal when creating the `beatle` tool.

3 Program

The simplest and most platform independent way to create visualization software is as a dynamic webpage. To build such a page, it is common to use a framework which reduces the complexity of writing pure HTML, CSS, and JavaScript significantly.

Having the possibility of creating interaction in the browser is only part of the equation, the second, more important part is the actual drawing of pixels to the screen which in this case is providing a user interface for displaying a BEATOR2 model. In this chapter these two issues will be discussed in depth including the challenges faced during implementation.

3.1 Environment

This subchapter serves as a very brief introduction to the programming language, framework and libraries used in creating the `beatle` tool.

Instead of directly writing HTML, CSS and JavaScript, a frontend web framework called *React* was employed. It is among the most used tools in this field and relies on JavaScript and specific syntax called *JSX* (JavaScript Syntax Extension) to create interactive web pages.

For initially constructing the webpage, a library called *create-react-app* was used. Like in most cases when developing for the web, *NodeJS* with the *Node Package Manager* (npm) was employed for local testing and installing of libraries.

With React interactivity, state and logic is directly implemented alongside the structure of the website. To facilitate this, custom HTML elements can be defined. These so-called components are functions that return a composition of other components and plain HTML elements. They can be either stateless or stateful, the latter requiring a concept called hooks to achieve this.

Instead of being called like other functions in JavaScript, the aforementioned JSX syntax is used. In Figure 4 a call to generate an `Interface` component can be seen. Creating a component via a normal function call is also valid but considered bad practice. Parameters, called *props* in this context, of a component are passed using attributes inside a tag.

```
// JSX call
<Interface model={processModel(text)} text={text} />

// equivalent standard JS function call
Interface({ model: processModel(text), text: text });
```

Figure 4: JSX call versus JavaScript function call.

Just as HTML is hierarchical, components follow the same principle. As a consequence, information can only be passed down the hierarchy and never up or to other components on the same level. This results in a predictable data flow, but also to data needing to be passed through multiple layers without being necessary in some.

JavaScript is a weakly typed language, meaning no explicit type annotations. Therefore, a problem that arises when creating complex applications is losing track of the allowed data types for a variable. If a problem occurs, it can only be caught at runtime, due to JS being an interpreted language. To alleviate this problem, a superset created by Microsoft called TypeScript was used. It introduces, among other things, proper type checking and safety when writing code.

Using TypeScript, the code is compiled to standard JavaScript. During this process, type mismatches and other syntactical problems can be caught and as a bonus, the IDE can use the type information to give proper autocompletion. The only slight disadvantage of using TypeScript is the increased verbosity of the code because variables and parameters are annotated with type information, which can, however, sometimes be inferred from their usage.

Using NodeJS for hosting the tool is one option. Unfortunately, this would require some server infrastructure to serve the webpage. Instead, it is hosted using GitHub pages. Utilizing the fact that the webpage is purely front end, static HTML, CSS and JavaScript files can be created. As the tool was developed using version control with a GitHub repository, this process was trivial using the `gh-pages` library.

3.1.1 Used Libraries

In addition to the necessary packages needed for React and Typescript, two different libraries were used in the process of creating this program. The first one, D3 (Data Driven Documents), is used for displaying the model as a graph, and the second, MUI, is used for styling the different elements of the interface.

When visualizing data D3 is the go-to JavaScript library to achieve interesting bar charts, pie charts, or in this case network graphs. It has a lot of capabilities and contains a lot of parts which are not necessary for this project. The part which is relevant, however, is the one used for creating so called force-graphs. In a later chapter the workings of it will be properly explained, and it is only mentioned here for completeness' sake.

Lamentably, it is not natively compatible with React and requires workarounds to be used inside the program. The part where the D3 library is necessary is considered as a black box for the rest of the program, and only the in- and outputs are perceivable from the other components in the hierarchy. This means this part of the software mostly was not written in TSX, the proper term for JSX using TypeScript, but in pure TypeScript using the classic approach of querying elements present on the webpage. This solution is not without its faults and the problems caused by this are described later in the chapter.

The MUI library provides predefined components for use in a React application. They are styled according to the defined theme and are quite varied in terms of options. The theme used for this program is Material UI. It follows the design philosophy of Google's material design and therefore reminds heavily of their products and the vanilla Android operating system.

Most importantly the library removed the necessity of creating a CSS file for styling the parts of the software. The only information needed to be provided when creating MUI components are the size and position of the element. The library also allows to modify the color scheme and other aspects of the theme to create more unique visuals. Figure 5 depicts the stylistic design of the buttons and the color layout. Instead of the standard blue accent color MUI provides per default, a more turquoise color was chosen.

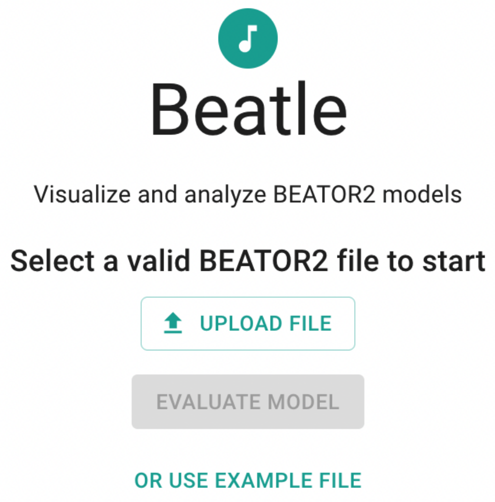

Beatle
Visualize and analyze BEATOR2 models
Select a valid BEATOR2 file to start
UPLOAD FILE
EVALUATE MODEL
OR USE EXAMPLE FILE

Figure 5: Landing page of the website

3.2 Parsing and Processing

The first step necessary for dissecting and displaying a model is to load it. For this tool, it can be done by either choosing to input a file via an HTML input form, or choosing a default pre-generated model file. Selecting either of these options only yields a model file as one singular string which is not very helpful without further processing.

In essence, the parser transforms every line in the input data, apart from comments and empty lines, into node representation. These nodes are of generic nature and are self-contained, meaning they save all necessary information acquired during this process and the ones described later internally. For this part specifically it keeps track of its class, sort, parents and depending on the class optionally an immediate or a string name.

Every line is split into these parts and via pattern matching a node object is generated. All generated nodes are recorded in a map with their ID as the key. The nodes corresponding to sorts and those actually part of the model are separately. These maps and the information that the currently parsed node can only reference already existing nodes lets us exploit the makeup of a line in the model file to just look up the parent nodes and directly save references inside the currently parsed node object.

One caveat are nodes of the class `init` because they do not represent a node in the graph but rather modify existing state nodes. These therefore are treated differently.

The output of the parsing procedure is not only the map of nodes and sorts but also arrays for the different kind of roots of the graph, meaning nodes of the classes `bad`, `next` and `state`. This saves unnecessary filter operations on the map and therefore computing time. Furthermore, other information about the loaded model is included. These will be mentioned in the following sections.

3.2.1 Local Calculations

The parsing of a model file is split into two parts. The first consists of the transformation of the nodes described before, in addition to calculations which will be described now.

One information that might be useful when analyzing a model is how dependent a node is on others, meaning how many nodes influence the evaluation of it. This number can be easily calculated in parallel to the transformation. A node, once parsed, never changes its attributes in this pass anymore. Therefore, the dependency is also fixed, and a node can easily just sum up the values of their parents for this metric plus the number of parents themselves.

However, two parents may depend on the same node and therefore a simple tally is not sufficient. Instead, references to the node objects are kept which are further split into their different classes. This results in the different class types being mapped onto arrays of dependencies. Unsurprisingly, this results in the properties that leaf nodes like constants and inputs always have no dependencies and on the contrary root nodes of the classes `bad` and `next` have the largest of a specific subgraph.

Another useful piece of information to calculate is the height, not to be confused with depth which will be described in the second parsing part. In this case the height means the distance to the furthest leaf. This information will be later used in determining the longest paths for each root node.

The way to calculate this number is simple. Leaf nodes know they are leaf nodes and therefore have a height of zero. The other nodes decide which parent has the larger height, add one to this value and store it as their own height. This means that once again the resulting value only depends on the direct parents and requires little computational costs.

The third and final local calculation is the only one done in the second part of parsing. It is the calculation of depth. Unfortunately, it cannot be easily derived from the height and therefore needs to be manually calculated.

In the program we differentiate between two different values for depth. One is the depth from a node of class bad or next, and the other is the depth from a state node. Obviously, not every node has a state node as one of its descendants and this is only conditionally calculated.

In both cases the strategies are identical. We start at the roots, which have depth zero, and recursively descend the hierarchy. The process is very similar to a depth first search and only pursues a path farther if the depth is larger than a previously calculated one. Unfortunately, doing one or two DFS for each node means the time complexity, unlike the first part, is not linear.

3.2.2 Global Calculations

Besides the local information calculated for each node, also some global attributes of the model can be gathered. These attributes fall into one of two categories. They can either be derived from the data before transformation or from the data calculated during the first and second parsing part.

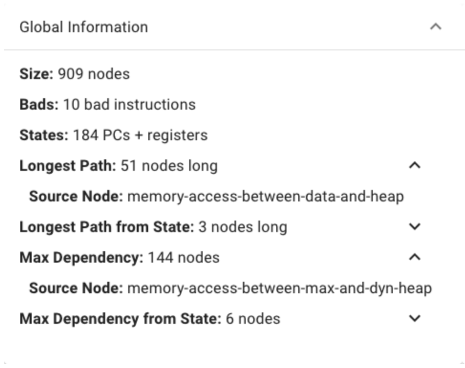
In the first case the maps and arrays of roots is taken, and their size is measured. This then gives us the information about the size, in terms of number of nodes, of the model and the number of the different kinds of roots in the model.

The second kind of information can easily be updated during the parsing parts. Internally a variable is updated every time a node would be the new candidate for a certain metric, and after the parsing is done the last candidate is the final value. For every metric of this kind, there is not only the value of it but also the source node responsible for producing this value.

The calculated metrics are the maximal dependency of the graph (starting from a state node), and the longest path in the graph (starting from a state node). Fortunately, we can once more exploit the knowledge that only root nodes may be responsible for producing these values.

3.2.3 Displaying the Calculations

In the user interface of the program two widgets are used to display the before described information. One locally for a singular node, the other for the global attributes of the model. How these look can be seen in Figure 6.


Global Information

Size: 909 nodes

Bads: 10 bad instructions

States: 184 PCs + registers

Longest Path: 51 nodes long

Source Node: memory-access-between-data-and-heap

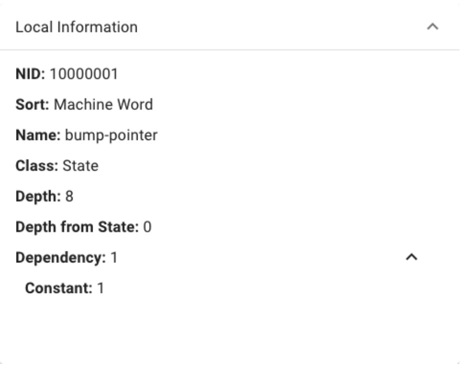
Longest Path from State: 3 nodes long

Max Dependency: 144 nodes

Source Node: memory-access-between-max-and-dyn-heap

Max Dependency from State: 6 nodes

(a) Global Information Widget


Local Information

NID: 10000001

Sort: Machine Word

Name: bump-pointer

Class: State

Depth: 8

Depth from State: 0

Dependency: 1

Constant: 1

(b) Local Information Widget

Figure 6: Widget for global and local metrics of the example model.

The karats in the heading of the widgets and besides certain metrics indicate, that this option can be collapsed or expanded. This means specifically that content of the widgets can be hidden and that for certain metrics like dependency further information is available.

In the global widget the source nodes in the additional information for metrics derived from the parsing parts are always referred to via their names. This is possible due to them always being either bad, next or state nodes which all have a string identifier for their name.

Another additional detail is the naming convention for the sorts. They are named differently as in the model file itself and there are fewer names used than defined in the file. Only four different sorts are displayed in the local information and later in the graph. These are a boolean type, a machine word, the virtual memory type and a sort called "Bytes" which represents a datatype which is one through seven bytes large and only used for modelling the `read()` system call, and each is used only once. Therefore, they can be grouped together.

3.3 Visualization

The information that is being calculated in the last subchapter does not necessitate a user interface and could have been done as a command line program if it were the only thing done for this thesis. Besides knowing information about a model, it is also important to be able to navigate paths and even cognitively imagine the look of a model.

The most natural approach suggesting itself by virtue of the structure of the model file is a tree or graph containing nodes and edges. Unfortunately, a tree falls out of the question, due to multiple nodes being able to reference the same node as a descendant. If this were possible the visualization of the model would be trivial and not as involved as the solution in this program. This subchapter explores the different flavors for displaying a graph in the program.

3.3.1 Displaying a Graph

The centerpiece of this thesis is the visualization of BEATOR2 model files in a directed acyclic graph. This consequently implies that one of the most important pieces in software is the presentation of nodes and data flow on a screen.

Due to the graphs being rather complex and containing a lot of vertices and edges it would not suffice to simply generate a static raster image that is drawn to the screen. Therefore, a technology is needed which provides scalability with regard to image size and additionally can be manipulated via code.

The natural choice is using an SVG for this purpose. Scalable vector images are defined similarly to HTML documents using tags and are natively supported in the browser. When creating an image there are different graphics elements you can choose from to display different shapes. The classic approach for displaying a graph is using circles for vertices in the graph and lines between the circles with arrow heads indicating the direction of the edges. Fortunately, all of these options are available as such graphics elements.

In figure Figure 7 an example for how the SVG in the actual graph looks like. To define an SVG you need to declare it with an `<svg>` tag. Additionally, a width and height of the image is provided. What follows is a group (`<g>` tag) for scaling and translating the canvas. Contained in it are two groups, one for the edges and the other for the vertices. Edges are realized with path elements and nodes with circles.

The `d` attribute provides the path for the element. The first *M* moves the starting position for the line to the provided coordinates and the *L* traces a line to the target destination. For the circles the coordinates of the center need to be provided as their own coordinate values (*cx* and *cy*). Additionally attributes for the radius and stroke width are included. The details omitted from the SVG code are the colors and the proper opening tag for a SVG.

```
<svg width="500" height="400">
 <g transform="translate(250,200)">
  <g stroke-width="10">
   <path d="M-150,0L150,0"/>
  </g>
  <g>
   <circle cy="0" r="20" cx="-150"/>
   <circle cy="0" r="20" cx="150"/>
  </g>
 </g>
</svg>
```

(a) Code for SVG

(b) Resulting SVG image

Figure 7: Simple SVG definition

Just as important as being able to display nodes and data flow is the generation of the layout of the graph. For this the force-directed graph part of the D3 library was used. With it the nodes can be given a charge that repulses other nodes near them. To disallow them to continuously drift apart they are chained together via their edges which can be adjusted to be more or less rigid. In addition to these two forces, two additional ones are used. One that attracts nodes to the center of the SVG canvas and the other that forbids nodes from overlapping.

Unfortunately, this approach does not minimize edge overlaps and does not construct very pleasant graph layouts. Nonetheless, this is the only computationally feasible approach for spreading nodes and edges.

The D3 library does all the heavy lifting for displaying the graph using the aforementioned force-layout. To do this, it is provided copious amounts of information. In addition to the node and edge data described later, it is given the following information. The size and color of the nodes depending on their class, the stroke weight for the edge depending on the needed sort, the necessary functions for the graph to be zoomable and singular nodes to be draggable, and the position of the arrow heads on the edge. The last needs to be properly calculated using geometric calculations, otherwise they would not be visible, due to the edges being from one center of a node to the other and not edge based.

Providing the edge and node information to the graph is also quite involved. Sadly, the map of the nodes calculated in the parsing and processing step cannot be directly used and need to be transformed first. D3 expects arrays for both the edges and nodes. For the latter, the currently displayed nodes can be directly pushed into an array and used as is. The edges on the other hand, need to be generated as objects. These contain the source and target nodes, and additionally the sort flowing over the edge.

To construct these, starting from the root node of the subgraph we generate an edge object from the currently targeted node and all of its currently relevant descendants. Finally, the edge weight is decided by the sort of the target node.

With all the pieces of information described so far the D3 library displays the desired subgraph in an SVG canvas. The graph itself has interactivity by the means of being able to drag nodes and zoom into the plane. How a subgraph may look like can be seen in Figure 8.

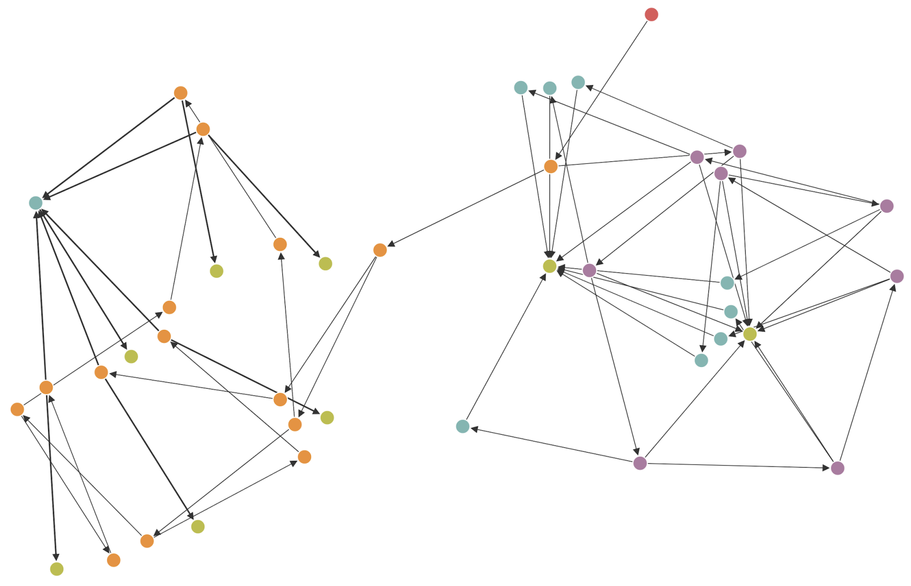
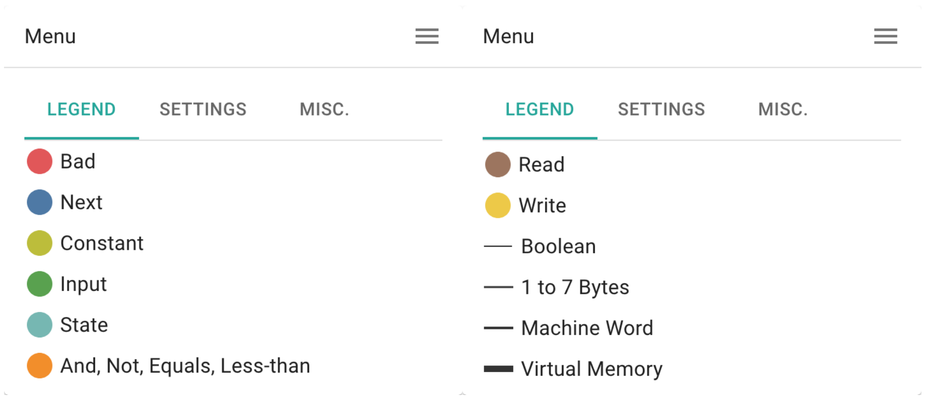


Figure 8: Subgraph for a bad node of the example model

As can already be seen, a user does not have enough information with the graph alone to know which colors correspond to what class and what stroke weight is used by what sort. Therefore, a legend exists in which this information is denoted. Due to not having infinite different colors at disposal, some classes are grouped together. If this is the case they correspond to the same type of operation like arithmetic or logical.

The legend is located in a widget different from the local and global information called *Menu* which also contains other tabs for other parts of the program. These will be described in later parts. In Figure 9 it can be seen how the legend looks like. A node, or edge is displayed in the same format as in the graph on the left-hand side, and what it corresponds to is written on the right.



(a) Top of the legend (b) Bottom of the legend

Figure 9: Two different sections of the legend

Due to the widget only being of limited size, not all information can be displayed at once, therefore a user needs to scroll to access the hidden information. Another additional measure for knowing which class a node has is the ability to hover over it. A tooltip is then shown displaying the class of the node.

When clicking on a node, it will be highlighted and the local information for it will be shown. Dragging a node will not automatically select it. This option is the only way to access the local information for a node.

3.3.2 Abstracting Complexity

Due to different subgraphs possibly containing a large amount of edges and nodes the graph quickly gets visually noisy and confusing. Additionally, the suboptimal way the layout is generated further increases the problem of being able to understand a model. The options of dragging nodes and zooming helps a little, but not enough. In this section two methods for reducing the number of edges and nodes in a graph will be described.

The first approach to abstracting complexity is hiding nodes in the hierarchy. A node in the subgraph can be either expanded or collapsed which means it's descendants are present in the current display of the subgraph or not, respectively. The exception to the rule are the leaf nodes, meaning constants and inputs because they do not have descendants.

The state a node is in can be toggled by clicking on it whilst it is highlighted, i.e. its local information is shown. Per default when first loading a model, the selected subgraph is fully expanded automatically. There is an option to disable this behavior, to allow navigating paths properly through the model. The feature itself does not need to be enabled and always works except in conjunction with the other abstraction approach.

The implementation for expanding and collapsing nodes is asymmetrical. For expanding the nodes only need to be added to the relevant node array if they are not already in it and the proper edges need to be constructed and added for the direct descendants.

Collapsing a node is more involved because it is not enough to just remove the direct children of a node, but also their children if there are no other nodes referencing them and so on. Therefore, this needs to be done in a recursive manner.

Figure 11 shows the exact snippet of code responsible for collapsing a node in the model. The code might look a bit intimidating, but the concept is simple. Remove the edge of the node from the graph and if a parent node has no edge to another node in the graph anymore recursively call the function on that node and then remove it from the graph.

```
const collapse = (n: ModelNode) => {
 n.collapsed = true;
 graphState.links = graphState.links.filter(l => l.source !== n);

 n.parents.forEach(x => {
  if(graphState.links.filter(l => l.target === x).length === 0){
   collapse(x);
   graphState.nodes.delete(x.nid);
  }
 });
};
```

Figure 10: Code for recursively collapsing a node.

An example of collapsing a node can be seen in figure Figure 11 in which collapsing the highlighted node does not lead to removing all its descendants.

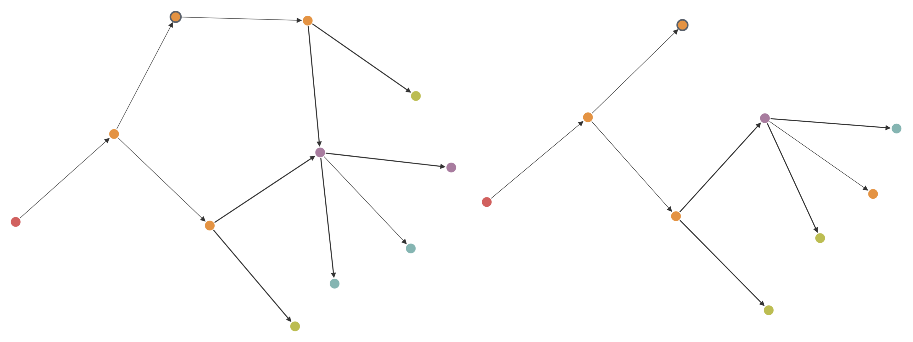
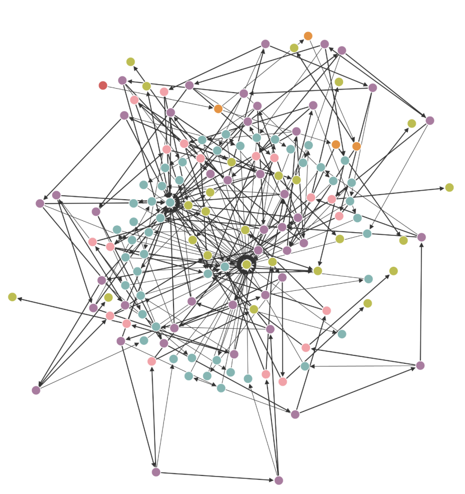

(a) Graph before collapsing (b) Graph after collapsing

Figure 11: Behavior of collapsing the highlighted node

This feature allows for a user to choose how much and what information of a subgraph to display at once. Unfortunately, with it a user cannot grasp the entirety of the subgraph anymore after collapsing some nodes while also reducing the complexity.

That is why the program also supports the second type of abstraction, called clumping. When using clumping, nodes of the same class or collection of similar classes can be grouped into one singular node. Depending on the distribution of the classes of nodes in a subgraph, this can lead to a drastic reduction in nodes and edges. It also means that a user can put his focus on particular classes of nodes not clumped. It is not necessary to always have every piece of information available at once, and if it were, there is still always the option of not using one of the abstraction features.

Clumping nodes in the graph leads to a new type of node being introduced. To visually differentiate a clump from its peers, the circle representing the node has a larger radius. This new kind of node in the graph also includes its own local information. It contains the class of node it abstracts, as well as the number of nodes and what the minimal and maximal depth of a node in the clump is. How clumping affects the complexity of a subgraph can be seen in Figure 12.


(a) Graph before clumping

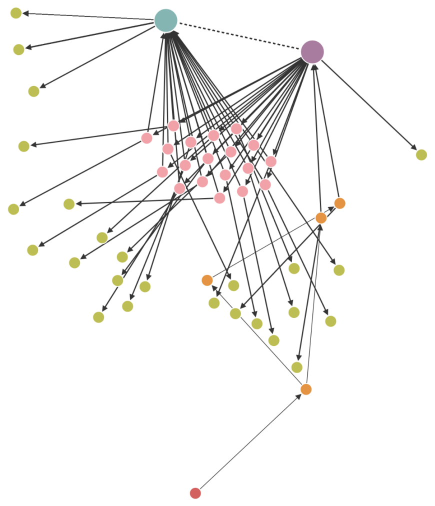

(b) Graph after clumping

Figure 12: Graph before and after clumping state (light blue) and if-then-else (lavender) nodes

In addition to the node type, a new type of edge is also introduced. While the flow of information into and out of a clump to normal nodes is not affected, it might happen that two clumps should have multiple connections between them meaning there would need to be multiple edges with possibly different directions and edge weights. Therefore, the program opts to use a dotted line if there is more than one edge connecting two clumps. This can also be seen in Figure 12.

As already mentioned, collapsing cannot be used together with clumping. Allowing both options for abstraction at once would lead to unintuitive interaction and behavior and would additionally increase the proper implementation complexity significantly.

3.3.3 Highlighting the Longest Paths

In the section about calculating global information it is already mentioned that the longest path for the entire model is recorded. Unfortunately, a user can only view what node is the root for the longest path. This is why there is also a setting to turn on the highlighting of the longest path, which can be seen in Figure 13.

The path is marked by the edges being colored red and being slightly thicker than normal. Additionally, it is not only possible to highlight the longest path for the global maxima, but also for each root that is selectable for a particular subgraph.

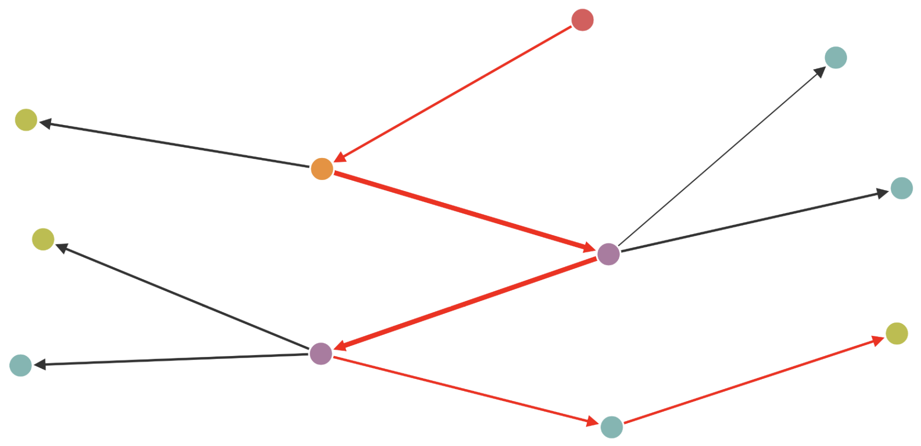


Figure 13: Longest path for a small subgraph rooted in a bad node

Contrary to the clumping feature, highlighting the longest path can be used at the same time as expanding and collapsing nodes. This permits a user to manually follow the longest path by disabling the automatic expansion of the graph. Additionally, the default behavior of the expansion is altered to not expand every node in the entire subgraph, but only the ones that are on the longest path.

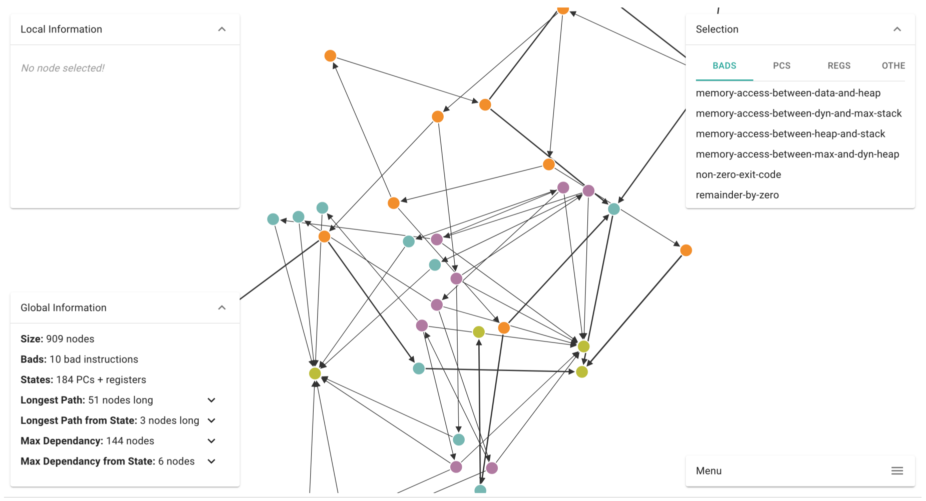
For the implementation of the highlighting and automatic expansion the already calculated value for the height is used. The longest path must always start at the root of the subgraph and its height is a certain value not equal to zero. For finding the next node on the path, the descendants are inspected to determine which one has a height that is one smaller than the parent.

It is possible that there are multiple possible candidates for the next node, i.e. there are multiple longest paths of the same length, but in this case the first node in the references to descendants in the node is picked as the next node on the path. This means that only one path is displayed, but it is always deterministically the same.

This construction of the path happens recursively and terminated when a node with a height of zero is reached, namely constant and input nodes.

3.4 Putting it all together

The features described in the prior subchapters do not exist independently but simultaneously in the same user interface. In Figure 14 the UI in its default state is shown. No node is selected for local information, the menu is collapsed, the root for the subgraph is the alphabetically first one and the subgraph is automatically expanded.


Local Information

No node selected!

Global Information

Size: 909 nodes
Bads: 10 bad instructions
States: 184 PCs + registers
Longest Path: 51 nodes long
Longest Path from State: 3 nodes long
Max Dependancy: 144 nodes
Max Dependancy from State: 6 nodes

Selection

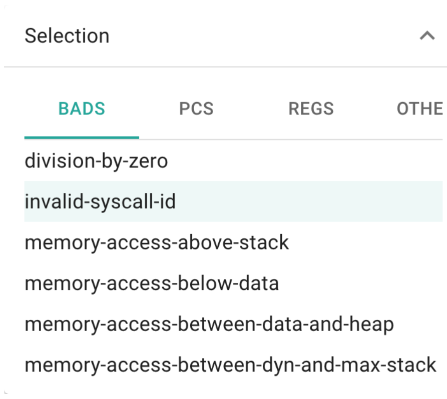
memory-access-between-data-and-heap
memory-access-between-dyn-and-max-stack
memory-access-between-heap-and-stack
memory-access-between-max-and-dyn-heap
non-zero-exit-code
remainder-by-zero

Figure 14: User interface directly after loading the example model file

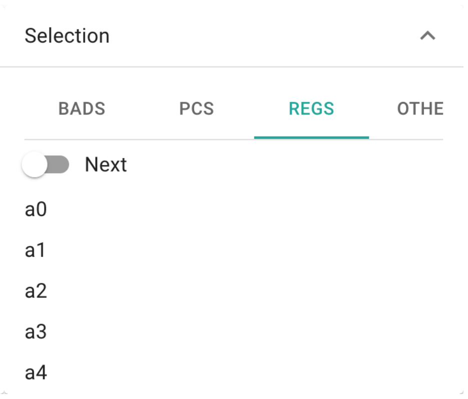
One detail that was omitted until now was the selection of a root for displaying a subgraph. As a reminder, a root is either a bad, next or state node. These represent (negated) safety properties, registers, program counter states and other states like kernel mode flags and the virtual memory.

For easier selection of the appropriate root, in the selection panels these roots are split into four different groups. The `PCs`, `Regs` and `Bad` groups are self-explanatory and the `Other` group contains the ones that do not fit into one of the other three. In the selection widget these groups are represented by tabs the content of which is sorted alphabetically. Every list element in a tab is clickable and upon doing so selects the element as the root for the subgraph.

Another important part is the switching between selecting next and state nodes. Every next node has a parent that is a state node, therefore both types of nodes are represented equally. To avoid redundancy in the tabs there is a toggle that changes the behavior of selecting a list element to be either a next or state node depending on what state the toggle is in. In Figure 15 content of two tabs for the example model can be seen.


Selection
BADS
PCS
REGS
OTHE
division-by-zero
invalid-syscall-id
memory-access-above-stack
memory-access-below-data
memory-access-between-data-and-heap
memory-access-between-dyn-and-max-stack

(a) Bads tab with higlighting of the selected element

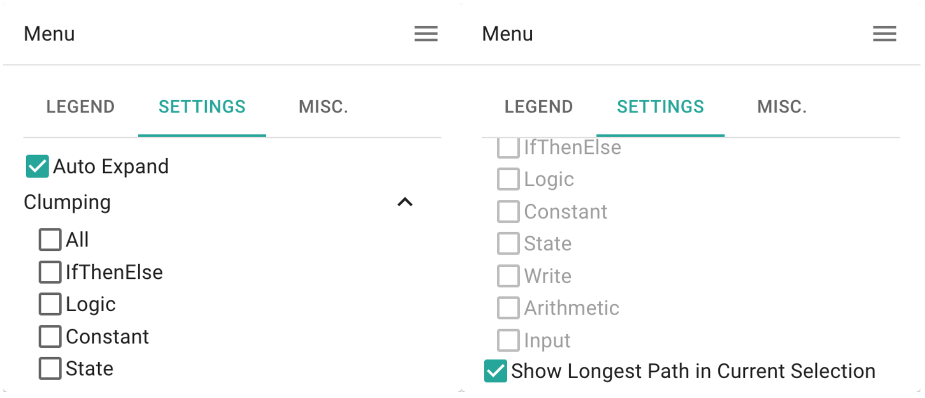


(b) Regs tab with next/state toggle set to state

Figure 15: Content of two tabs of the selection widget

As already mentioned in different subchapters, there are some settings which can be changed for certain features. These are in all cases to turn them off or on. The way to change these settings are in the `Settings` tab in the menu widget. The settings available can be seen in Figure 16. They are always checkboxes and are comprised of disabling the automatic expansion of a subgraph when selecting a node, turning clumping on or off for different node types and whether to turn on the highlighting of the longest path.

Per default the only option turned on is the automatic expansion and due to their nature they always cause the subgraph to reload. The clumping settings can be collapsed and there is a button to activate all clumping settings. As can also be seen in the figure, the program automatically disables the possibility of toggling certain features when some checkmarks are set. These are the disabling of clumping when the longest path is highlighted, and forcefully enabling the automatic expansion and disabling path highlighting settings when any clumping is active.



[✓] Auto Expand
Clumping
[ ] All
[ ] IfThenElse
[ ] Logic
[ ] Constant
[ ] State

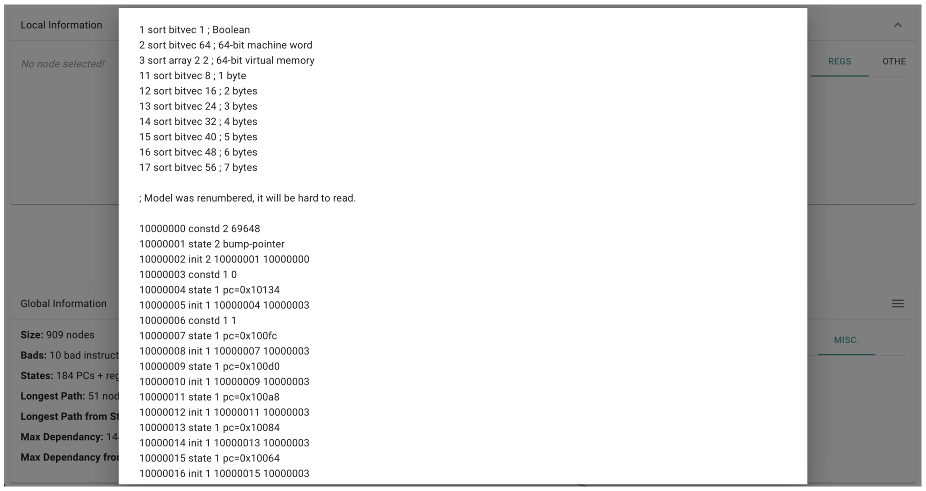
[ ] IfThenElse
[ ] Logic
[ ] Constant
[ ] State
[ ] Write
[ ] Arithmetic
[ ] Input
[✓] Show Longest Path in Current Selection

(a) Top of settings tab with default options. Clumping settings can be collapsed

(b) Clumping options grayed out due to showing the longest path

Figure 16: Available settings for different features

The final feature not described so far is the insight into what model is actually loaded. This is especially helpful when viewing the example model file. In the menu widget there is a third tab in addition to the legend and the settings, called `Misc` which contains for one a link to this thesis and a button for showing the currently loaded model in a popup modal. A user can scroll in it and when they are finished just click outside it to hide the file.



Size: 909 nodes
Bads: 10 bad instruct
States: 184 PCs + reg
Longest Path: 51 nod
Longest Path from St
Max Dependancy: 14
Max Dependancy fro

```
1 sort bitvec 1 ; Boolean
2 sort bitvec 64 ; 64-bit machine word
3 sort array 2 2 ; 64-bit virtual memory
11 sort bitvec 8 ; 1 byte
12 sort bitvec 16 ; 2 bytes
13 sort bitvec 24 ; 3 bytes
14 sort bitvec 32 ; 4 bytes
15 sort bitvec 40 ; 5 bytes
16 sort bitvec 48 ; 6 bytes
17 sort bitvec 56 ; 7 bytes

; Model was renumbered, it will be hard to read.

10000000 constd 2 69648
10000001 state 2 bump-pointer
10000002 init 2 10000001 10000000
10000003 constd 1 0
10000004 state 1 pc=0x10134
10000005 init 1 10000004 10000003
10000006 constd 1 1
10000007 state 1 pc=0x100fc
10000008 init 1 10000007 10000003
10000009 state 1 pc=0x100d0
10000010 init 1 10000009 10000003
10000011 state 1 pc=0x100a8
10000012 init 1 10000011 10000003
10000013 state 1 pc=0x10084
10000014 init 1 10000013 10000003
10000015 state 1 pc=0x10064
10000016 init 1 10000015 10000003
```

Figure 17: Modal for showing the model file

3.5 Known Issues

The larger a graph, the more difficult it is to reasonably display. D3 starts to struggle to simultaneously display more than 100 node large graphs on slower hardware. Even the simplest of source binaries may result in models with at least a thousand nodes. This means that displaying the entire graph at once is out of the question.

Performance is the big issue which prevents this software from being scalable to very complex models. The constraint of only showing a subgraph, as well as the expanding/collapsing and clumping features help to lessen the problem. Still, when first loading a model these settings are not enabled and therefore the performance tanks. The MUI library is another culprit of introducing performance impact by having some overhead for their components.

The D3 library brings with it some other flaws as well. Regretfully, it is the only viable option to use for this application, barring implementing the logic from scratch which would be for one very time-consuming to implement and for another would probably perform even worse. One major flaw is that the documentation for the different features and options in the force graph are all over the place and due to large changes in newer versions of the library most old tutorials are heavily outdated and therefore very unhelpful. Moreover, the official documentation exists in a monolithic Markdown file and is incomplete at the time of writing this thesis.

One of the most infuriating issues is the layout creation of the force-graph. There are no options, that would not impact performance even more, which would help reduce the amount of edge crossings in a graph leading to more pleasant visuals. There might be charge values for nodes and edges which help, but these can only be found via trial and error.

Another issue with D3 is the integration into React. Every time the user expands or collapses a node, selects a new root node, or changes a setting the entire graph re-renders. Unfortunately this is unavoidable due to the quirky interaction between what causes a React component to re-render and the workings of D3. This means a user can easily lose track of the position of the node of interest. This is very jarring and lessens the user experience.

Another issue of the program and the format it is presented in, i.e. a website, are the limited input capabilities when using a cursor. The only available mouse button is the left click. The others are reserved for other operations in the browser which could be overridden, but this leads to accessibility problems. Only having the left click leads to clunky interactions between dragging, highlighting and trying to collapse or expand a node.

4 Conclusion and Outlook

Conclusively, the `beatle` tool helps with visually displaying the BEATOR2 model format, but not without its issues. A complex graph can easily be overwhelming. Even though the abstraction opportunities help reduce the problem, they do not fully mitigate it. Nonetheless, the tool helps to visualize and analyze moderately sized models. Even during writing it has proven to be helpful in grasping the meaning of different parts in different model files.

There is currently the effort to further increase the prowess of the BEATOR2 format by extending the modeling capabilities to a more complete set of the RISC-V instruction set. This would most likely necessitate employing more operations from the source BTOR2 format rendering the current implementation of the tool obsolete for these new type of models. This is not a disheartening piece of information but reminds that this thesis only provides a snapshot of the current state of the `beatle` tool, which will most likely grow alongside the changes of the model format.

Further adding to the possibilities of change in the tool are extensions of the functionality. These may include but are not limited to moving to the third dimension to reduce the amount of edge crossings, that the validity of the model file is validated and the possibility of viewing actual information flow by providing the opportunity to provide actual inputs for evaluation. The latter two are already features of the `BEATOR` tool implying that this would mean that the two tools grow closer together and may even merge into one.

5 References

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[2] Edmund M Clarke. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[3] Christoph M. Kirsch and Stefanie Muroya Lei. Quantum advantage for all, 2021.

[4] C.M. Kirsch. Selfie projecct. `https://github.com/cksystemsteaching/selfie`. [Accessed 3-July-2022].

[5] C.M. Kirsch and M. Starzinger. Symbolic execution versus bounded model checking. In *Invited paper to appear in Festschrift 2022*. Springer, 2022.

[6] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, btormc and boolector 3.0. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 587–595, Cham, 2018. Springer International Publishing.