INSTITUT FÜR INFORMATIK
LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Improving the Encoding of Arrays in Btor2-to-C Translation

Salih Ates

Bachelor's Thesis

Author:	Salih Ates
Supervisor:	Prof. Dr. Dirk Beyer
Mentors:	Nian-Ze Lee
	Po-Chun Chien
Submission Date:	September 20, 2023

Declaration of Authorship

Hereby, I declare that I have composed the presented bachelor's thesis independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

Munich, September 20, 2023 Salih Ates

Acknowledgments

I would like to express my sincere gratitude to my mentors, Nian-Ze Lee and Po-Chun Chien, for their invaluable guidance and support throughout my thesis work. They have been instrumental in helping me develop and refine solutions for the challenges presented in this thesis work and have always been quick to assist me with any issues that arose during my research. Their assistance was invaluable from the perspective of a bachelor's student.

I would also like to thank the Software and Computational Systems Lab for providing me with access to the VCloud services, which enabled me to conduct large-scale experiments in a seamless way. The VCloud services were a powerful platform for software verification, and I am grateful for having been able to use them for my research.

Abstract

Model checking and verification are important tasks in ensuring the correctness and reliability of hardware and software systems. They aim to prove or disprove that a system meets certain specifications or properties that are desired or required for its functionality. However, these tasks can be challenging and time-consuming due to the complexity and size of the systems and their models. Therefore, various tools and techniques have been developed to facilitate and automate model checking and verification processes. Btor2C is one such tool that translates word-level sequential circuits in the hardware model Btor2 format into behaviorally equivalent C programs, enabling software verifiers to handle hardware verification tasks. However, software verifiers face a challenge when dealing with C programs translated from Btor2 circuits involving array sorts, which are commonly used to model memories in hardware designs. None of the evaluated software verifiers could prove any of the C programs translated from these Btor2 verification tasks, indicating a weakness of software verifiers in dealing with arrays. To address this issue, we have developed two solutions that aim to improve the performance of software verifiers on Btor2 circuits with arrays. The first solution blasts the arrays in a Btor2 circuit into sequences of bit-vectors and simulates operations on arrays with bit-vector operations. The solution is implemented as a standalone script and can be used as a preprocessor before Btor2C. The second solution is implemented as an enhancement to Btor2C, which aims to create a heuristic ordering of intermediate circuit nodes by scheduling writing operations to arrays as late as possible. Through this targeted heuristic scheduling, we can minimize the number of duplicates needed for writing operations. We evaluate both solutions on a benchmark set that consists of Btor2 tasks, collected from various sources such as the Hardware Model Checking Competitions, and compare them with state-of-the-art hardware and software verifiers. Our results show that the blasting approach has effectively increased the number of verified Btor2 tasks with arrays overall while the second solution provided more insights into the patterns in which write operations appear within these hardware models. The work presented in this paper demonstrates the potential of preprocessing data to achieve greater performances of tools and also presents further research ideas and potential solutions for issues that were identified during this work.

Contents

1 Introduction

Hardware and software systems are ubiquitous in modern society, and their correctness and reliability are crucial for various applications and domains. However, ensuring that these systems meet certain specifications or properties that are desired or required for their functionality can be challenging and time-consuming due to the complexity and size of the systems and their models. Therefore, formal verification and testing are important tasks that aim to prove or disprove the correctness of hardware and software systems using rigorous mathematical methods and tools. In the field of formal verification and testing, hardware and software systems are often analyzed by different tools and methods, despite sharing common theoretical foundations and solving techniques. To benefit from the advancements of both communities, Prof. Dr. Dirk Beyer, Nian-Ze Lee, and Po-Chun Chien proposed Btor2C [10]. This is a tool that translates word-level sequential circuits in Btor2 [6] format into C programs. Btor2 is a common format for hardware verification, and C is a widely used language for software analysis. With this open-source tool, software verifiers can complement hardware verifiers by solving some of the Btor2 hardware models that were previously unsolved by hardware verifiers. The results of the paper on Btor2C have shown that the translation tool achieved its goal of removing barriers for formal verification. However, these experiments have also shown that software verifiers had difficulties proving those Btor2 translated tasks that featured arrays. Out of 157 tasks that contained array sorts in their models, none of the software verifiers that were used in the experiments were able to provide proofs within the given resource limits.

1.1 Motivation and Contribution

The difficulty of software verifiers in handling arrays in Btor2 translated tasks motivates us to find possible solutions for this problem. Arrays are a common and useful data structure in both hardware and software domains, and they often represent memory or storage components. In this work, we propose two solutions to improve the Btor2-to-C translation for tasks with arrays. The first solution is implemented as a standalone script that gets rid of the arrays completely by making use of the blasting technique, which separates the bit-vector elements of an array into their own bit-vector variables. The second solution is implemented as an enhancement to Btor2C by adding an option

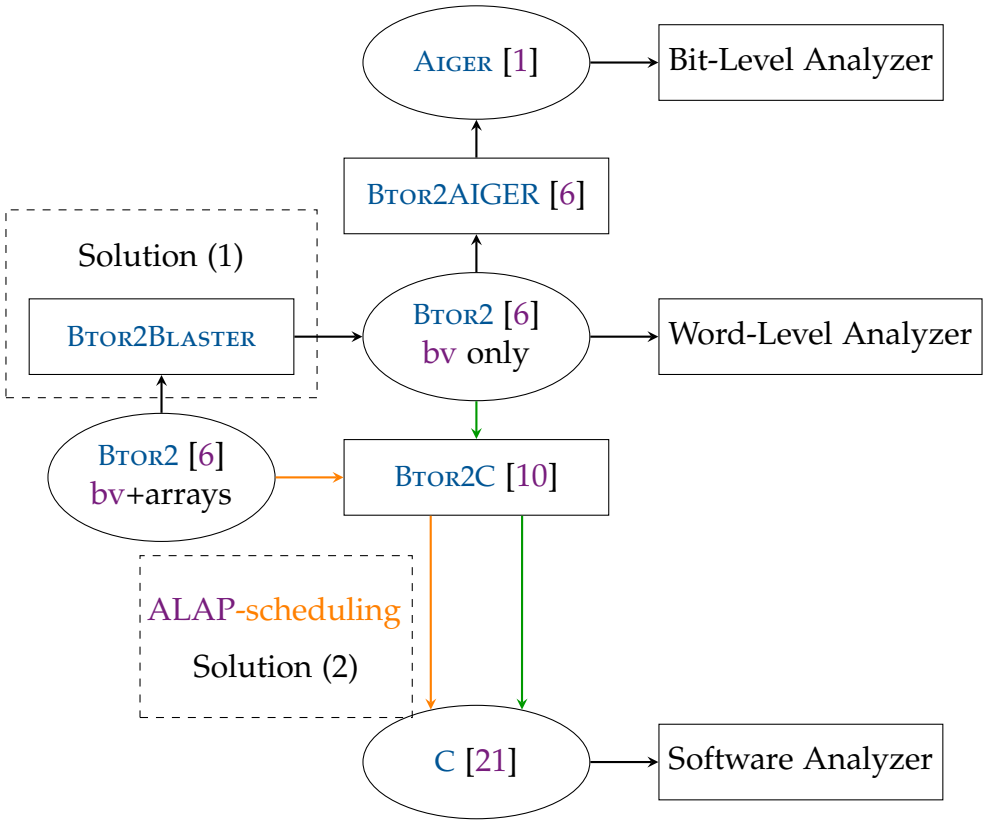

Figure 1: Our solutions in the BTOR2 translation flow

that reduces the number of array duplications in the translated C programs. The option applies a heuristic scheduling algorithm that delays write operations on arrays as much as possible in the translation schedule. Without enabling this option, Btor2C creates a separate new array for each write operation on an array.

Our main contributions are:

1. Replace Arrays by Sequences of Bit-Vectors

We present a blasting approach that eliminates arrays from Btor2 circuits by transforming them into sequences of bit-vectors. This approach enables software verifiers to handle Btor2 tasks with arrays more effectively. We show that our blasting approach helps software verifiers increase their overall number of proofs from 0 to 7. On top of that, hardware verifiers benefitted from our solution as well. Btor2Blaster enables the use of the bit-level analyzer ABC [24] on actual Btor2 array tasks which was not possible before because Btor2AIGER [6] can only translate Btor2 models which do not feature array variables. The number of solved tasks for hardware verifiers increased from 129 to 139.

2. Reduce the number of Arrays by avoiding unnecessary Duplications

We present a heuristic scheduling approach that reduces the number of array duplications in the translated C programs by delaying write operations on arrays as much as possible. This approach aims to reduce the memory consumption and complexity of the translated C programs, which could potentially improve the performance and scalability of software verifiers. However, enabling this option in Btor2C did not result in a significant increase in the number of solved tasks by software verifiers. Instead, it revealed some interesting patterns and challenges in the Btor2 tasks with arrays, which suggest some directions for future work and improvement.

How both solution fit into the translation flow is illustrated in Figure 1. We evaluate both approaches on a benchmark set that consists of 318 Btor2 tasks with arrays, collected from various sources such as the Hardware Model Checking Competitions [2]. In addition to that, we also manually created a set of 36 tasks as a separate benchmark set for our evaluation. We compare our solutions with state-of-the-art hardware and software verifiers, and analyze the results in terms of correctness and performance.

```
1 sort bitvec 2
2 sort bitvec 4
3 sort array 1 2
4 state 3
5 constd 1 2
6 constd 2 4
7 write 3 4 5 6
8 input 1
9 read 2 7 8
10 sort bitvec 1
11 eq 10 9 6
12 bad 11
```

(a) Btor2 example with array

```
1 sort bitvec 2
2 sort bitvec 4
3 sort bitvec 1
4 constd 1 0
5 constd 1 1
6 constd 1 2
7 constd 1 3
8 state 2
9 state 2
10 state 2
11 state 2
12 constd 1 2
13 constd 2 4
14 input 1
15 eq 3 4 14
16 eq 3 5 14
17 eq 3 6 14
18 ite 2 17 13 11
19 ite 2 16 9 18
20 ite 2 15 8 19
21 sort bitvec 1
22 eq 21 20 13
23 bad 21
```

(b) Btor2 array blasted

Figure 2: An example Btor2 circuit with an array (a) and its blasted equivalent (b)

1.2 Examples

1.2.1 Replacing Arrays by Blasting

We illustrate the principle behind the blasting approach with an example Btor2 circuit in Figure 2. The Btor2 format is a common format for representing word-level sequential circuits, which are widely used in hardware verification. The Btor2 format uses a line-based syntax, where each line defines a node in the circuit. A node can be a sort declaration, an input or state variable, an operation, a constraint, or a property. The command `state` is used to define bit-vector or array variables, in the example's case we define an array sort of size 4. The information about the size of the array sort can be read from the sort definition in line 3. We explain how this is done in later sections of this thesis. The Btor2Blaster splits this array state into a sequence of separate bit-vector states. A bit-vector is created for each possible index value. In Figure 4b, line 8 corresponds to the first index and the following bit-vector states correspond to the remaining indexes. This is the core idea that we implement throughout all array operations. Chapter 4 describes how we translate each of the operations into a blasted equivalent.

```c
for (;;) {
  // Getting external input values ...
  // Assuming invariants ...
  // Asserting properties ...
  SORT_2* var_7_arg_0 = state_4;
  SORT_1 var_7_arg_1 = var_5;
  SORT_2 var_7_arg_2 = var_6;
  SORT_3 var_7;
  for (unsigned char i = 0; i < (1 << 2); ++i){
    var_7[i] = var_7_arg_0[i];
  }
  var_7[(unsigned char) var_7_arg_1] = var_7_arg_2;
  // Computing next states ...
  // Assigning next states ...
}
```

(a) C program

```c
for (;;) {
    // Getting external input values ...
    // Creating intermediate signals ...
    SORT_2* var_7_arg_0 = state_4;
    SORT_1 var_7_arg_1 = var_5;
    SORT_2 var_7_arg_2 = var_6;
    SORT_2* var_7 = var_7_arg_0;
    var_7[(unsigned char) var_7_arg_1] = var_7_arg_2;
    // Assuming invariants ...
    // Asserting properties ...
    // Assigning next states ...
}
```

(b) C program with as-late-as-possible (ALAP)-Scheduling enabled

Figure 3: Write operation done without (a) and with ALAP-Scheduling enabled (b) (write only for demonstration purposes)

1.2.2 Reducing the Number of Arrays

Btor2C is a tool that takes a Btor2 circuit file as input and outputs a C program file that is behaviorally equivalent to the circuit. The C program consists of three main parts: the declarations, the initialization, and the next-state function. The declarations part defines the variables and types that correspond to the nodes and sorts in the circuit. The initialization part assigns the initial values to the variables that correspond to the state nodes in the circuit. The next-state function updates the values of the variables according to the operations and dependencies in the circuit. The next-state function is executed repeatedly in an infinite for loop until a property node is violated or a constraint node is unsatisfied. This simulates the sequential behavior of the circuit.

The goal of the second solution is to reduce the number of array duplication by

scheduling write operations to arrays as late as possible. Array duplication occurs when Btor2C creates a new array variable for each write operation on an array and copies all the elements from the original array variable. This consumes memory and increases the complexity of the C program, which makes it harder to analyze by software verifiers. By scheduling write operations as late as possible, we can increase the number of times where we reuse the original array variable and update it directly without creating a new array variable. This can only be done where it is confirmed that the correctness of the translation is not affected. The condition requires that the needed information is never lost i.e., the overwritten value is not needed after the writing process.

To achieve this goal, we have implemented an option for Btor2C that performs ALAP-scheduling for write operations on arrays. ALAP-scheduling works by finding a topological order of the Btor2 circuit nodes that tries to increase the number of avoided duplicates for write operations. By default Btor2C does not really follow any particular order but translates nodes recursively to not violate data dependencies. Our algorithm tries to find an ordering by delaying write operations as much as possible, while respecting the dependencies and properties of the circuit. However, our algorithm does not guarantee an optimal ordering, since it does not account for special cases that can occur. Therefore, our algorithm is a heuristic that aims for a near-optimal ordering that improves the performance and scalability of software verifiers on Btor2 translated tasks with arrays.

We illustrate this approach with the translation of the write operation from the previous Btor2 example in Figure 3. Without the ALAP-scheduling option, Btor2C creates a new array variable for each write operation and copies all the elements from the original array state (lines 8-11 in Figure 3a). This results in unnecessary duplication and memory consumption. With the ALAP-scheduling option, Btor2C reuses the original array state and updates it directly without creating new array states (lines 7 and 8 in Figure 3b). This avoids unnecessary duplication and memory consumption.

2 Related Work

2.1 Blasting Data

One of the existing tools for translating Btor2 models is Btor2AIGER [6], which converts Btor2 files into the bit-level format Aiger by bit-blasting bit-vectors. The idea behind Btor2AIGER is to apply bit-blasting to each bit-vector expression in the Btor2 input and generate an equivalent Aiger output. This method can preserve the semantics of the original model, but can also introduce a significant blowup depending on the bit-vector widths.

This line of work is related to our Btor2Blaster tool in that it applies similar ideas. Our approach differs from Btor2AIGER in several aspects. First, we blast arrays instead of bit-vectors. This can be seen as a generalization of the bit-blasting technique used by Btor2AIGER, as we apply it to arrays only and use bit-vectors instead of single bits to preserve the semantics of the model. Second, we do not translate the Btor2 model into another format but rather replace the array variables in the file itself. This can be seen as a preprocessing step because our initial goal is to improve the performance of various verifiers on these array tasks. Btor2AIGER bridges the gap between these word and bit-level formats.

Btor2Blaster can be seen as an extension that helps bridge the gap between Btor2 and Aiger formats by enabling Btor2AIGER to translate array models. Since Btor2AIGER cannot translate array tasks in its current state, our tool is of great benefit in this field. This way, our tool can facilitate the use of bit-level analyzers that take Aiger format files as input on blasted array models.

2.2 Preprocessing to Boost Performance

Another related work is sQueezeBF [19], which is an effective preprocessor for quantified boolean formulas (QBFs). QBFs are an extension of propositional logic that allows variables to be universally or existentially quantified. QBFs are more expressive and compact than propositional formulas, but also more difficult to solve. Therefore, various tools and techniques have been developed to facilitate and automate QBF solving.

sQueezeBF is a preprocessor that applies various techniques for eliminating variables

and clauses from QBFs, such as variable elimination by Q-resolution, equivalence substitution, and equivalence breaking. These techniques aim to reduce the size and complexity of the QBFs, making them easier to solve by QBF solvers. The experimental analysis shows that sQueezeBF can produce significant reductions in the number of clauses and variables, and can improve the efficiency of a range of state-of-the-art QBF solvers.

Our technique of array blasting is similar to sQueezeBF in the sense that both aim to preprocess data in order to improve the performance of various tools. However, there are some differences between them. First, array blasting operates on hardware models in BTOR2 format, while sQueezeBF operates on QBFs. Second, array blasting eliminates arrays from the models and replaces them with bit-vector operations, while sQueezeBF eliminates variables and clauses from the QBFs. Third, array blasting can be used as a preprocessor before BTOR2C or as a standalone tool, while sQueezeBF is a preprocessor for QBF solvers.

In summary, our technique of array blasting is related to some previous work that also aims to preprocess data in order to improve the performance of various tools. However, our techniques have some distinctive features and advantages that make them novel and valuable contributions to the field of formal verification.

3 Background

3.1 Hardware Model Checking

The reachability safety problem for hardware asks whether a sequential circuit always satisfies a given safety property. The safety property is usually specified by an output signal of the circuit, called bad. A sequential circuit has a combinational part that performs computations and memory elements that store the circuit state. The circuit operates in discrete time frames, and in each time frame, the combinational part takes the current state and the external input as inputs, and produces the output and the next state as outputs. A hardware model checker can solve the reachability safety problem by checking if there is any input sequence that can make the circuit produce a bad output. If such an input sequence exists, the circuit is unsafe and the model checker will report it. Otherwise the circuit is considered as safe [10].

3.2 Software Model Checking

Software model checking is a way of verifying if a program meets a given specification. The specification is often expressed as an error location that should not be reached by any execution of the program. Software model checking is generally harder than hardware model checking, because software programs can have unbounded behaviors and data structures. However, many methods have been developed to overcome this difficulty [25], such as predicate abstraction [26, 28], counterexample-guided abstraction refinement [17], and interpolation [27, 22]. These methods, together with the advances in SMT solving [8], enable the verification of large-scale software systems [10]. Btor2C made it possible that these methods can be applied to hardware models.

3.3 The Word-Level Model Checking Format Btor2

Btor2 [6] is a format for modeling word-level sequential circuits. Sequential circuits are circuits that use memory elements to store and use previous state information to determine their next state, unlike combinational circuits, which only depend on the

current input values to produce outputs. The Btor2 language model is often used because of its simplicity in providing sufficient operations over bit-vector and array sorts. It is supported by many hardware model checkers and also the input format in competitions like the Hardware Model Checking Competitions (HWMCC) [2]. Compared to Verilog [20], another hardware description language with a rather complicated syntax, Btor2 provides a perfect platform for working as an intermediate representation for hardware models. Yosys [13] can translate Verilog models into behaviorally equivalent and simpler Btor2 models, which is another reason why Btor2 is the perfect language for this intermediation task. Btor2 is a language that is supported by many hardware model checkers and, thanks to Btor2C, can also be used with software analyzers that take C files as their input. Btor2 was developed with the idea of being a generalization of the bit-level Aiger format [1]. Refer to the original Btor2 publication [6] for more details on the syntax.

In this thesis, we explain only the necessary information about the syntax. A Btor2 line usually starts with a unique number that serves as either a sort or node identifier for the line's entity, which can then be used as an argument by other operations in case of node identifiers (first number in each line of Figure 2a are not line numberings but unique sort or node identifiers). This format follows a topological order in which arguments used by an operation must be defined before the execution of the actual operation itself. After the identifier of an entity, an operation name follows, which is then followed by a sort identifier that specifies the sort of the entity. All identifiers after this sort id are node identifiers and are used as arguments for the operation. The exceptions are bad state properties and constraints (invariants), which take a single node identifier as an argument. Signatures of Btor2 operators generally follow this signature: `<node id> <op> <sort id0> <node id1> [<node id2 [node id3]>]`. Later operations that use the result of `op` must use `id` as an argument. We can initialize a state with the `init` construct and specify the transitional behavior of a state with `next`. The following section explains how we can translate this format into a C program.

3.4 Translating Btor2 Models into C Programs

Btor2C [10] is a tool that translates Btor2 hardware models into behaviorally equivalent C programs. Numerous software analyzers that take C programs as inputs can now, thanks to Btor2C, be applied to actual hardware models and thus barriers between these two verifier fields have been removed. The tool generally translates each line into its own variable `var_<id>` with the exception of lines that are `sort` definitions, `state`, `input`, `init`, `next`, and `bad`, in which case it follows this pattern: `<op>_id`. In the actual creation of the translated program, the tool follows a pattern where it first defines

the sorts, then creates and initializes the states, and then enters the actual sequential circuit as shown in Figure 12. The circuit behavior is simulated by an infinite for-loop and inside that loop, the tool first checks the invariants (`constraint` in BTOR2), then checks the safety of the properties, and then assigns the next values to the states for the following loop. All of these processes of initializing, checking invariants and safety properties, and assigning next states are done by recursively creating all the nodes that they need in order to be executed. In this thesis, we introduce another template for the creation of nodes. For better visualization and more details on the translation process, please refer to the original publication about BTOR2C [10]. We address the actual problems that have arisen for tasks that contain arrays in the following chapters.

4 Blasting Arrays into Bit-Vectors

Arrays in Btor2 are a convenient way to model memories that can store and retrieve values at arbitrary addresses. However, for analysis purposes, it might be useful to represent arrays in a more explicit and concrete way, using only bit-vectors. Bit-vectors are fixed-length sequences of bits that can be manipulated by bitwise operations. By knowing the bit-widths of the array index and element bit-vectors, we can blast arrays into separate bit-vectors, one for each possible index-value pair. This technique is called array blasting and it allows us to access or update any element by simply selecting or modifying the corresponding bit-vector.

In Figure 4a, we give a circuit whose state is an array of sort 3 (line 4). This sort uses a bit-vector of width 3 to address an index and stores elements of sort 2, which are bit-vectors with a width of 8 (lines 1-3). The bit-width of the index bit-vector provides the necessary information about the actual number of elements that can be stored in it. Sizes of arrays are always powers of two; in case of the example in Figure 4a, this size equates to 8. With the necessary information provided about how many elements (and their sorts) can be stored in the array, we know the exact number of needed state bit-vectors. The array state gets translated into a sequence of separate element bit-vectors in Figure 4b, where the first state bit-vector corresponds to the first index element, the second to the second, and so on (lines 3-10). We can access or modify elements at specific indexes within this blasted state by applying these operations to the corresponding bit-vectors.

This blasting approach requires an adaptation of all the operations that can use arrays as arguments. These are: `write`, `read`, `eq`, `neq`, `ite`, `init`, and `next`. `write` and `read` are the only ones that are array exclusive and thus require more effort for a proper translation. Thanks to Btor2's great number of operations that it offers, we can simulate them by simple ways such that we preserve the original semantics.

The transformation process works line by line and thus the first array related operation always is the definition of an array sort. Because we intend to replace all arrays, definitions of array sorts are not required anymore. We replace this definition by another definition of a bit-vector sort with the width 1 followed by constant decimals (constd) that correspond to the index numbers. We do this replacement because later operations require the determination of index values, and by doing this, we avoid a constant repetition of this block.

```
1 sort bitvec 3
2 sort bitvec 8
3 sort array 1 2
4 state 3
```

(a) Btor2 circuit with an array

```
1 sort bitvec 3
2 sort bitvec 8
3 state 2
4 state 2
5 state 2
6 state 2
7 state 2
8 state 2
9 state 2
10 state 2
```

(b) Btor2 array blasted

Figure 4: An example Btor2 circuit with an array (a) and its blasted equivalent (b)

```
1 sort bitvec 2
2 sort bitvec 8
3 sort array 1 2
4 state 3
5 constd 1 2
6 constd 2 9
7 write 3 4 5 6
8 read 2 7 5
9 add 2 8 6
```

(a) Write with constant as index

```
1 sort bitvec 2
2 sort bitvec 8
3 sort bitvec 1
4 constd 1 0
5 constd 1 1
6 constd 1 2
7 constd 1 3
8 state 2
9 state 2
10 state 2
11 state 2
12 constd 1 2
13 constd 2 9
14 add 2 13 13
```

(b) Blasted write with constant as index

Figure 5: Blasting write operations with constant index bit-vector

4.1 Adapting Array Exclusive Operations

Along with `read`, `write` is one of the most commonly used array operations in Btor2 models. Write operations in Btor2 have the following signature: `<node id> write <sort id3> <old_array id3> <index id2> <element id1>`. The operation takes three arguments: an array node, an index node, and an element node. It returns a new array node that is equal to the original except at the given index where it has the given value.

One of the advantages of the blasting approach is that it can simplify the operations on arrays when the index bit-vector is a constant. In such cases, the corresponding bit-vectors that represent the value stored at the index can be directly accessed and manipulated. Later operations that would use these values as arguments would now use the id of the corresponding bit-vectors directly. In Figure 5a, you can see that in line 7, a write operation is done at index with the id 5, which is a constant that corresponds

```
1 sort bitvec 2
2 sort bitvec 8
3 sort array 1 2
4 state 3
5 input 1
6 constd 2 9
7 write 3 4 5 6
```

(a) Write with `input` as index

```
1 sort bitvec 2
2 sort bitvec 8
3 sort bitvec 1
4 constd 1 0
5 constd 1 1
6 constd 1 2
7 constd 1 3
8 state 2
9 state 2
10 state 2
11 state 2
12 input 1 index
13 constd 2 9 value
14 eq 3 4 12
15 eq 3 5 12
16 eq 3 6 12
17 eq 3 7 12
18 ite 2 14 13 8
19 ite 2 15 13 9
20 ite 2 16 13 10
21 ite 2 17 13 11
```

(b) Blasted write with `input` as index

Figure 6: Blasting write operations with arbitrary index bit-vector

to the decimal number of 2. The value read in line 8 is then used for an add operation. In the blasted example (Figure 5b), the bit-vectors in lines 8-11 correspond to the blasted array variable in line 4 of the original state. Knowing that a write operation is done at index 2, we can keep using the old array's bit-vectors with the exception at the written index, in which we now use the id of the written value. In this example, the new array created by the write operation corresponds to lines 8 (index 0), 9 (index 1), 13 (new written value at index 2), and 11 (index 3).

Not always are constants used as the index bit-vector whose values can be directly read in the transforming process. Constant variables in the BTOR2 format are the only variables whose values can be directly read from the line since the value appears as an argument. In case of every other operation, this is not the case, which makes a general solution necessary to determine the index value. The solution to this problem can be solved by making use of ite (if-then-else) operations. The signature is as follows: `<nid> ite <sid> <condition id> <bitvec id1> <bitvec id2>`. We make use of ite statements in order to return the right bit-vector at each index of the the new written version of the array. The blasting process is shown in Figure 6. First, the conditions are prepared (lines 14-17), which will then be used in the ite operations. As mentioned before, the array sort definition part (line 3 in Figure 6a) is replaced with a bit-vector

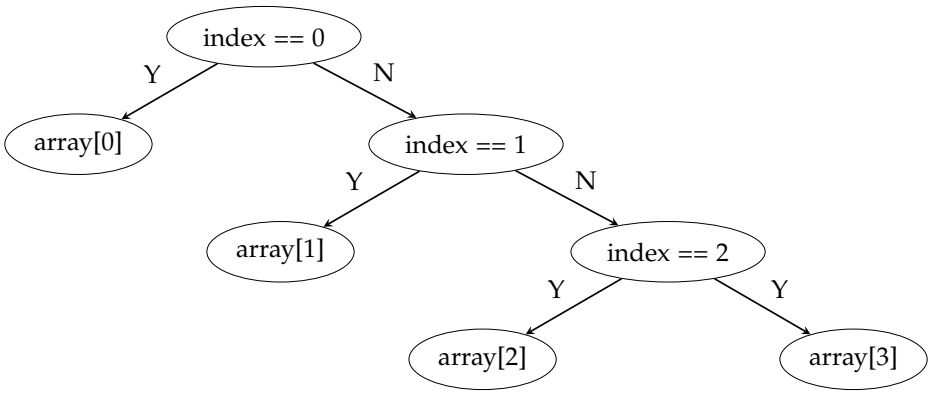

Figure 7: Determining the index by using a skewed ite tree

width 1 sort and constants (lines 3-7 in Figure 6b). When preparing conditions, we make use of these constants and the sort. They are used in lines 14-17 due to the reason that eq needs to be of a bit-vector sort with the bit-width of 1 and the need of constant values to compare the index to so that the index value can be determined. With the conditions prepared, the following lines with the id 18-21 represent the newly written array's indexes from 0 to 4. At each index of this newly overwritten array is an ite operation that returns either the id of the bit-vector that was intended to be written into the array or the id of the old array's bit-vector. In line with the id 18, the condition with the id 14 checks if the index bit-vector is equal to the constant bit-vector of the same sort with a value that represents the decimal number 0. If this condition holds, the bit-vector with the id 13 is returned which is the constant value that was being written into the array in the non blasted file. Otherwise, the bit-vector at index 0 of the old array (line 8) is being returned. This way, the operations with the id 18-21 represent the indexes of the newly written version of the old array.

The other array exclusive operation is read. The signature of this operation is as follows: `<nid> read <sid1> <array id> <index id1>`, which returns the value of the provided array at the given index.

Just like with the write operation, cases of arbitrary index bit-vectors is also given for read operations. This problem can again be solved by making use of Btor2's ite operation. Difference here is that a read operation returns a single bit-vector, which is why ite statements are chained up. There are multiple ways of determining the correct index and returning the single bit-vector at the found index. In this thesis, two ways of chaining ite operations are implemented.

```
1 sort bitvec 2
2 sort bitvec 8
3 sort array 1 2
4 state 3
5 input 1
6 read 2 4 5
```

(a) Btor2 with read operation

```
1 sort bitvec 2
2 sort bitvec 8
3 sort bitvec 1
4 constd 1 0
5 constd 1 1
6 constd 1 2
7 constd 1 3
8 state 2
9 state 2
10 state 2
11 state 2
12 input 1
13 eq 3 4 12
14 eq 3 5 12
15 eq 3 6 12
16 ite 2 15 10 11
17 ite 2 14 9 16
18 ite 2 13 8 17
```

(b) Skewed blasted Btor2 example of the read operation

Figure 8: Blasting read operations with arbitrary index bit-vector

4.1.1 Skewed Read

The first method determines and returns the bit-vector at the given index by chaining ite operations into a skewed tree. This is the default option for the blasting script. A determination is being done by comparing the index bit-vector with each possible index number that is possible. The first step is to check whether or not the index is equal to 0 and based on the result either the id of the bit-vector at the given index of the array is returned or the next ite operation in which the index is compared to the next index number. This continues until the index is determined. In Figure 8b, this is done in lines 16-18 and the process visualized in Figure 7. Since Btor2 follows a topological order, the arguments of an ite statement must already exist. Due to this, the ite operations are created in a bottom-up manner when looking at Figure 7. Later operations that would use this read value would refer to the bit-vector with the id 18 in the given example in Figure 8b. This method can lead to a deep nesting of if-then-else statements in the translated C program and in order to lessen the possible depth, an additional way of reading from blasted arrays has been implemented.

4.1.2 Balanced Read

The script offers another option that can be enabled with the -b option and changes the way that read operations are done on blasted arrays. Activating this option creates a

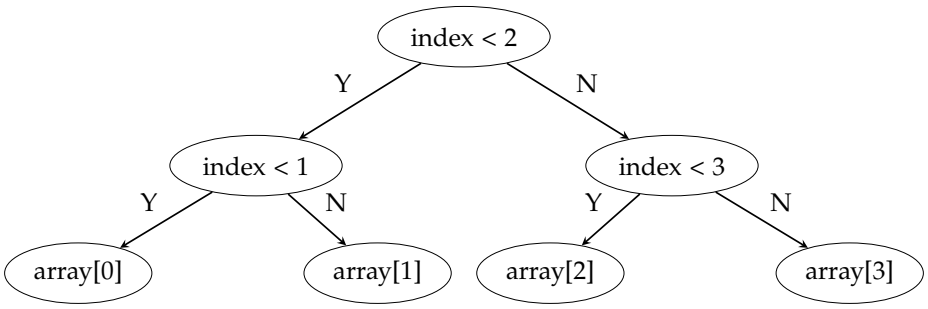

Figure 9: Determining the index by using a balanced ite tree

```
6 state 2
7 state 2
8 state 2
9 state 2
```

(a) Blasted array(1)

```
10 state 2
11 state 2
12 state 2
13 state 2
```

(b) Blasted array(2)

```
15 (init/next) 2 6 10
16 (init/next) 2 7 11
17 (init/next) 2 8 12
18 (init/next) 2 8 13
```

(c) Blasted init/next operation

```
15 eq 1 4 5
16 ite 2 15 6 10
17 ite 2 15 7 11
18 ite 2 15 8 12
19 ite 2 15 9 12
```

(d) Blasted ite operation

```
15 (eq/neq) 1 6 10
16 (eq/neq) 1 7 11
17 (eq/neq) 1 8 12
18 (eq/neq) 1 9 13
19 and 1 15 16
20 and 1 19 17
21 and 1 20 18
```

(e) Blasted eq/neq operation

Figure 10: Blasted array operations applied on each array bit-vector element

balanced tree instead of a skewed tree for determining the right index and the value at the given index. This approach is visualized in Figure 9. As described at the beginning of this chapter, the sizes of arrays are always powers of two, which is why the tree ends up being balanced in any case. This method of reading has a time complexity of $\mathcal{O}(\log n)$ with a lesser depth in nesting of ite operations, compared to the skewed's approach.

4.2 Adapting Non-Array Exclusive Operations

The remaining operations eq, neq, ite, init, and next all follow a similar structure when getting blasted. They apply operations on each bit-vector of both arrays which correspond to the same index in their respective arrays. Init and next are operations that assign values to existing variables and do not return or are not getting accessed. Instead of initialising an array or assigning the state of another array, we now have to

assign or initialise each bit-vector of the blasted array. Every bit-vector must correspond to the same index. In Figures 10a and 10b, two arrays are represented in their blasted states. A simplified example of how initialising or assigning next values is done with blasted arrays is shown in Figure 10c. BTOR2's ite operation returns an array based on a condition, which now in the blasted state returns the correct bit-vector at each index (based on the same condition) as it is shown in Figure 10d. The blasted bit-vectors of the array returned from this ite operation correspond to lines 16-19. Of these access operations, the only ones that require additional steps are the eq and neq operations. The comparison of each bit-vector is again done bit-vector by bit-vector (Figure 10e lines 15-18) but the actual confirmation that both arrays are either equal or not equal is done by taking the conjunction of all eq or neq operations (lines 19-21).

4.3 Limitations

Having implemented a blasting method for every operation that can take arrays as arguments, a transformation of BTOR2 models with arrays into bit-vector only equivalents can be executed. A drawback that comes with the array blasting technique is that it can produce very large files when given arrays of grand size. Arrays that use index bit-vectors of width 25 can store more than 30 million bit-vectors. Blasting an array of this size leads to the creation of BTOR2 format files with a minimum of 2^{25} lines. With additional array operations like read or write, these can result in files with hundreds of millions of lines and thus produce very large files. This is a disadvantage that comes at the cost of ridding BTOR2 files of arrays.

5 As-Late-As-Possible Scheduling to Reduce Array Duplications for Write Operations

A way of reducing the overall number of arrays by creating a schedule using the ALAP-scheduling algorithm is explained in this section. We explain how our algorithm works and why it is correct. We also discuss the complexity of our algorithm. Since Btor2 does not support mutable arrays, every write operation requires creating a new copy of the array with the updated element. This can result in a large number of arrays in the translated files, which can have an impact on the performance of the software analyzers that process them. Our solution is to optimize the schedule of the write operations by scheduling them as late as possible. The idea is to keep the old array without copying it when performing a write operation, as long as the old array is not accessed after the write operation. This way, we can avoid creating unnecessary array copies that are never accessed after write operations. In Figure 11, a Btor2 circuit with three write operations is shown. All writes happen in a sequence where they take the previous write array as an argument to write into except for the first one which takes the array state with the id 3 as its argument. Without ALAP-scheduling, we would create four separate arrays in the C translated program. Our algorithm reduces this to a single array in this example, because for all write operations, overwriting the old array does not result in information loss.

Our main goal is to reduce the number of arrays in the translated files, and thus improve the efficiency of the software analyzers. Our hypothesis is that our algorithm can preserve the semantics of the original source code.

5.1 Algorithm Description

Our solution is based on the principle of ALAP-scheduling, which delays write operations until they are required to be scheduled. It takes as input a set of circuit nodes that represent a Btor2 model, and a parser that can access them. A schedule is returned, which is a vector of node ids that specifies the order in which the nodes should be translated into C code. Our algorithm consists of two main steps: (1) finding the ready nodes, which are nodes that have no unscheduled fan-in nodes left; and (2) scheduling the ready nodes and 'visiting' their fan-outs while doing so. We use two stacks to store

Algorithm 1 ALAP-Scheduling of write operations

```
1: function CreateSchedule(circuit_nodes, parser)
2:     schedule ← []
3:     ready ← []
4:     ready_write ← []
5:     for i ∈ [0, circuit_nodes.max_id + 1] do
6:         if circuit_nodes[i].fanouts then
7:             line_tag ← circuit_nodes[i].line.tag
8:             if line_tag ∈ [state, input, const, constd, consth, zero, one, ones] then
9:                 ready.append(circuit_nodes[i])
10:            end if
11:        end if
12:    end for
13:    while ready or ready_write do
14:        if ready then
15:            popped_node ← ready.pop()
16:        else
17:            popped_node ← ready_write.pop()
18:        end if
19:        schedule.append(popped_node.line.id)
20:        popped_node.scheduled ← True
21:        if popped_node.tag == write and popped_node.needs_copy() then
22:            popped_node.duplicate ← True
23:        end if
24:        for fanout_node in popped_node.fanouts do
25:            fanout_node.visits+ = 1
26:            if fanout_node.visits == fanout_node.nargs then
27:                if fanout_node.tag == write then
28:                    ready_write.append(fanout_node)
29:                else
30:                    ready.append(fanout_node)
31:                end if
32:            end if
33:        end for
34:    end while
35:    return schedule
36: end function
```

```
1 sort bitvec 10
2 sort array 1 1
3 state 2
4 input 1
5 constd 1 0
6 constd 1 1
7 constd 1 2
8 write 2 3 4 5
9 write 2 8 6 7
10 write 2 9 4 6
11 read 1 10 4
12 sort bitvec 1
13 eq 12 11 5
14 bad 13
```

Figure 11: A Btor2 circuit with three write operations

the ready nodes: one for non-write nodes, and one for write nodes. We give priority to non-write nodes over write nodes, because we want to delay a write operations as much as possible. We use two fields to keep track of each node's status: *visits* and *scheduled*. The *visits* field counts how many arguments of a node have been scheduled (or by how many fan-ins they have been 'visited'); *scheduled* indicates whether a node has been added to the schedule or not. We use another field to indicate whether a node needs to copy its old array or not: *duplicate*. *duplicate* is true for a write node if its old array is accessed after the write operation; *duplicate* is false otherwise. We initialize our algorithm by finding all state, input, or constant nodes that have fan-outs, and adding them to the ready stack. These are nodes that have no predecessors, and thus are always ready. We repeat our algorithm until both stacks are empty. In each iteration, we pop a node from one of the stacks, add it to the schedule, and mark it as scheduled. If it is a write node, we check if the array, that the write node has as its argument, has any other fan-out that has not been scheduled yet. If all other fan-outs of the old array have been scheduled, then it is confirmed that there is no access operation done on this array after the writing process. This provides us with the opportunity of avoiding a duplication of the array. After popping a node, we update the *visits* field of all its fan-outs, and add them to one of the stacks if they become ready (have all their arguments scheduled). Pseudocode of this implementation is shown in Algorithm 1.

5.2 A new Template for C Translated Programs

In Section 3.4, we explained what template is being followed in the translation process from Btor2 to C. A new template was created specifically to facilitate this algorithm (Figure 12). Unlike the default translation where nodes are created recursively when

```
1  void main() {
2    // define sorts and constants
3    typedef ... SORT_a;
4    const SORT_a const_b = ...;
5    // initialize states
6    SORT_a state_c = const_b;
7    for (;;) {
8      // assume constraints
9      var_d = ...;
10     assume(constraint_e);
11     // assert properties
12     var_h = ...;
13     assert(!bad_f);
14     // compute next states
15     var_i = ...;
16     // update states
17     state_c = next_g;
18   }
19 }
```

(a) Old generic BTOR2C translation template

```
1  void main() {
2    // define sorts and constants
3    typedef ... SORT_a;
4    const SORT_a const_b = ...;
5    // initialize states
6    SORT_a state_c = const_b;
7    for (;;) {
8      // compute intermediate signals based on the schedule
9      var_d = ...;
10     // compute next states
11     var_i = ...;
12     // assume constraints
13     assume(constraint_e);
14     // assert properties
15     assert(!bad_f);
16     // update states
17     state_c = next_g;
18   }
19 }
```

(b) New BTOR2C translation template for ALAP-scheduling

Figure 12: Translation templates differ between default and ALAP option

they are needed for an operation, ALAP-scheduling creates an order that should be followed in order to reduce duplications. In this new template shown in Figure 12b, we do not separate intermediate signals like it is done by the recursive method. Instead, all intermediate signals are created before any checks and assignments are done. By doing this, we make use of the freedom in that we can order nodes as we want as long as the topological order which BTOR2 follows is respected.

5.3 Algorithm Correctness

Theorem 1. *The ALAP-scheduling algorithm preserves the topological order of the circuit nodes and does not introduce any semantic errors due to avoidance of array duplications.*

Proof. We prove the theorem by showing two properties of the algorithm:

1. The algorithm schedules each node only once, and only when all its fan-ins have been scheduled. This ensures that the topological order of the circuit nodes is respected, and that no data dependencies are violated. Moreover, the algorithm gives priority to non-write nodes over write nodes, which means that write nodes are delayed as long as possible.

2. The algorithm checks if a write node needs to copy its old array or not, by making use of another function (needs_copy() in Algorithm 1). This function returns true if and only if the old array has any other fan-out that has not been scheduled yet. This would mean that the old array is still accessed after the write operation, and thus it should not be overwritten. Therefore, our implementation avoids creating unnecessary array copies only when no loss of information is confirmed.

From these two properties, we can conclude that our algorithm does not introduce any semantic errors due to array duplication. This is because our algorithm at no point violates the topological order and ensures that duplicates are only avoided if information cannot be lost. ALAP-scheduling keeps the semantics intact of the original BTOR2 task. □

5.4 Algorithm Complexity

Calculating the time complexity is a good way of assessing the effectiveness of an algorithm. We assume that the input size is n, which is the number of circuit nodes in the BTOR2 circuit. We also know that the function to determine whether or not a copy is needed for a write has a linear time complexity. The algorithm consists of two main steps: finding the ready nodes, and scheduling the ready nodes.

The first step involves iterating over all the circuit nodes and adding them to one of the stacks if they have no left fan-ins, that are unscheduled. This takes $\mathcal{O}(n)$ time, since we visit each node once. The second step involves popping a node from one of the stacks, adding it to the schedule, and updating the *visits* field of its fan-outs. This takes $\mathcal{O}(m)$ time, where m is the number of fan-outs of a node.

For each node, we visit all of its fan-outs. Therefore, the total time complexity of our algorithm is $\mathcal{O}(n \times m)$. In a worst case scenario, all nodes could have $n - 1$ fan-outs. This would make our algorithm have a quadratic time complexity, which is less efficient for large BTOR2 models with such cases. However, in practice such situations are very rare.

6 Evaluation

To demonstrate that our proposed contributions in Chapters 4 and 5 enhance the current Btor2-to-C translation flow, we followed the same experimental setup as the original Btor2C publication [10]. Claims stated in Chapter 1 will be evaluated by answering following questions:

- **RQ1**: Can array blasting increase the number of solved array tasks?
- **RQ2**: Does using the ALAP-scheduling option of Btor2C increase the number of solved array tasks?
- **RQ3**: Do the implemented solutions complement each other?

6.1 Benchmark Set

The benchmark set that is being used for this thesis consists of various hardware verification tasks in Btor2 format. These were collected by the authors of Btor2C. They obtained these tasks from various sources, such as previous hardware verification competitions [12], projects, repositories and their own contributions. The reproduction package of this paper contains the complete list of sources and the whole benchmark set. In addition, a set of 36 Btor2 tasks were manually created and will be used for experiments separately. The contributions presented in this paper are mainly focused on improving array translation. Therefore, we excluded all tasks that do not have any array sorts. This reduces the number of tasks to 318 out of the total 1912. The collection of these real case tasks has 276 safe, 24 unsafe and 18 unknown tasks, of which the verdict yet remains to be determined. All tasks in the manually created set are safe.

Due to Btor2C's limitation in that it can only translate tasks that do not feature bit-vectors with a bit-width greater than 128 bits, we were able to translate 175 (154 safe, 21 unsafe) out of the 318 hardware models into C programs. This number is greater than that of the paper for Btor2C in which it was 157 translated array tasks. The difference comes from the fact that during the publication, Btor2C only supported bit-vectors up to a bit-width of 64 bits. In order to prove the presented solutions effectiveness, we made use of all tasks that we could run experiments on.

Thanks to the blasting method, we were able to array blast all Btor2 hardware models and got 318 array-less equivalents. Although that now arrays are not featured anymore, Btor2AIGER [6] still has limitations in that it cannot bit-blast those tasks that have non-constant initializations of variables. With this limitation, we were able to translate only 51 (27 safe, 24 unsafe) out of the 318 actual array tasks into Aiger format files. At this point, the presented solution of array blasting has already proven to be useful in that we can now use any array tasks at all with ABC.

All tasks are given to each tool in the required format.

6.2 Analyzers

An appropriate evaluation can be done by making use of state-of-the-art hardware and software analyzers. Since this thesis aims to highlight the improvements of both solutions, we made use of the same exact tools used in the original paper except for one analyzer. In order to be faithful to the original paper, we also made use of the same configurations.

Hardware Model Checkers

To compare hardware analyzers with software analyzers on actual hardware model tasks, the developers of Btor2C selected ABC [24] (at commit `a9237f5`[1]) and AVR [5] version 2.1 for hardware analysis. For the evaluation, we made use of the exact same versions. ABC is a bit-level model checker and takes Aiger format files as input. In this case, we end up using ABC on 51 tasks in total. AVR is a word-level hardware model checker that won HWMCC 2020 [2]. This word-level model checker can be used with Btor2 hardware model tasks directly. For both verifiers, property directed reachability (PDR)[18] is used as the verification algorithm. During experiments, we noted that *k*-induction [4] had a greater number of proofs and alarms for AVR but since a direct comparison is aspired to, we kept PDR as the solving algorithm.

Software Analyzers

These hardware model checkers then were compared with the software analyzers CPAchecker [14], Esbmc [23] and veriabs [7] in the original publication. All verifiers are state-of-the-art verifiers that occupied high ranking spots in recent competitions e.g., SV-COMP 2022 [9] (*ReachSafety*). Due to licensing issues, veriabs was replaced with Cbmc [16]. All of the software verifiers were obtained using the archiving repository

[1] https://github.com/berkeley-abc/abc

for the SV-COMP 2023 except for CPAchecker, for which a more recent version (2.2.1) was used, to check recent additions of configurations. CPAchecker will solve tasks using predicate abstraction, Esbmc *k*-induction, and Cbmc bounded model checking [11].

6.3 Experimental Setup

To stay faithful to the original circumstances, we set up an identical experimental enviroment. All experiments were ran on Ubuntu 22.04 (64 bit) running systems, with each system using a 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM. The limitations for each verification process are also identical with 2 CPU cores, a CPU-timelimit of 15 minutes and 15 GB of RAM. Version 3.14 of BenchExec[2] [15] was used to ensure reliable resource measurement and reproducible results.

6.4 Results

To compare the performance of the verifiers and the effects of the implemented solutions, all results are summarized in the tables 1-4. Displayed are the number of both correctly and wrongly solved tasks for each tool with every configuration used. Software analyzers are using four different configurations, '/' is used for the default translation of Btor2C. Tasks that were only translated with the lazy modulo option of Btor2C and none other fall into this category. We used the lazy modulo option because experiments in the publication of Btor2C showed that this option generally provides better results. Balanced and skewed configurations are the tasks that were translated from the blasted Btor2 files that were either created with the balanced or skewed option. Tasks for the ALAP configuration were translated with the ALAP-scheduling and the lazy modulo option enabled. AVR uses the default and blasted configurations while ABC can only be used on the blasted Btor2 models.

RQ1: Solving Blasted Btor2 Tasks

The results of the verifiers on the blasted configurations are summarized in Tables 1-4. We observe that array blasting has a significant impact on the performance of both hardware and software verifiers. In particular, we note the following findings:

[2] https://github.com/sosy-lab/benchexec

Tool	ABC		AVR		
Algorithm	PDR		PDR		
# Tasks	51		318		
Input	AIGER		BTOR2		
Configuration	balanced	skewed	/	balanced	skewed
Proofs	11	11	129	**137**	136
Alarms	**5**	**5**	0	2	2
Wrong proofs	0	0	0	0	0
Wrong alarms	0	0	0	0	0
Timeouts	35	35	85	97	98
Out of memory	0	0	0	0	0
Other inconclusive	0	0	104	82	82

Table 1: Summary of the results for hardware verifiers on the benchmark collection used for BTOR2C

- ABC is able to verify any array tasks at all only thanks to our array blasting solution. Without blasting, ABC cannot handle array operations in BTOR2 models. With blasting, ABC becomes the hardware verifier that finds the most alarms (5) on the original benchmark set, and solves 33 out of 36 tasks on the manually created set.
- AVR also benefits from array blasting, especially from the balanced option. Balanced blasting increases the number of proofs from 129 to 137 on the original benchmark set, and also enables AVR to find any correct alarms at all (2). On the manually created set, AVR increases its number of solved tasks from 24 to 32 with balanced blasting.
- CPACHECKER is the software verifier that benefits the most from array blasting. Without blasting, CPACHECKER cannot provide any proofs at all on the original benchmark set. With blasting, CPACHECKER achieves the most correct proofs (6) among the software verifiers, using the skewed option. On the manually created set, CPACHECKER increases its number of solved tasks from 4 to 15 with skewed blasting.
- ESBMC does not benefit from array blasting and produces the same results as in the original paper. This suggests that ESBMC is not able to handle bit-vector operations efficiently.
- CBMC suffers from a decrease in performance due to array blasting. On the original benchmark set, CBMC drops from 19 to 15 solved tasks with blasting. On the manually created set, CBMC cannot solve any tasks with any configuration.

Tool	CPAchecker				Esbmc				Cbmc			
Algorithm	Pred. Abs.				k-Induction				BMC			
# Tasks	175				175				175			
Input	C				C				C			
Configuration	/	balanced	skewed	ALAP	/	balanced	skewed	ALAP	/	balanced	skewed	ALAP
Proofs	0	4	**6**	0	0	0	0	0	0	0	0	0
Alarms	0	0	0	0	2	2	2	2	19	15	15	**20**
Wrong proofs	0	0	0	0	0	0	0	0	0	0	0	0
Wrong alarms	0	0	0	0	0	0	0	0	0	0	0	0
Timeouts	173	167	165	173	68	48	48	69	0	0	0	0
Out of memory	0	1	1	0	105	125	125	104	92	87	86	92
Other inconclusive	2	3	3	2	0	0	0	0	64	73	74	63

Table 2: Summary of the results for software verifiers on the benchmark collection used for Btor2C

This is because all tasks are safe and use infinite loops, which Cbmc cannot terminate within the given time and memory limits.

These results show that array blasting is a valuable technique for enhancing the verification of hardware models by both hardware and software analyzers. By transforming array operations into bit-vector operations, we enable more verifiers to handle array tasks and increase their number of solved tasks. However, not all verifiers benefit from blasting equally, and some may even perform worse. Therefore, choosing an appropriate blasting option and verifier is crucial for achieving optimal verification results.

RQ2: Solving Tasks Translated with the ALAP-Wcheduling Option

The second research question aims to investigate whether using the ALAP-scheduling option of Btor2C can increase the number of solved array tasks by the software verifiers.

- None of the software verifiers were able to take advantage of the ALAP-scheduling option on the original benchmark set (Table 2). This is because all write operations in 174 out of 175 tasks require copies of the old arrays, which prevented our algorithm from avoiding duplicates. This is due to a certain pattern of write operations that appears in most of the tasks, which we will discuss in more detail in Section 7.1 and propose a solution for future work in Chapter 7.
- On the manually created set, which was designed to avoid the pattern problem, most software verifiers saw improvements thanks to the ALAP-scheduling option. CPAchecker benefited the most from this option, as it increased its number of solved tasks from 4 to 25, which is the highest among all configurations. Esbmc

Tool	ABC		AVR		
Algorithm	PDR		PDR		
# Tasks	36		36		
Input	AIGER		BTOR2		
Configuration	balanced	skewed	/	balanced	skewed
Proofs	33	33	24	32	27
Alarms	0	0	0	0	0
Wrong proofs	0	0	0	0	0
Wrong alarms	0	0	0	0	0
Timeouts	3	3	0	4	5
Out of memory	0	0	0	0	0
Other inconclusive	0	0	12	0	4

Table 3: Summary of the results for hardware verifiers on manually created tasks

Tool	CPACHECKER				ESBMC				CBMC			
Algorithm	Pred. Abs.				*k*-induction				BMC			
# Tasks	36				36				36			
Input	C				C				C			
Configuration	/	balanced	skewed	ALAP	/	balanced	skewed	ALAP	/	balanced	skewed	ALAP
Proofs	4	14	15	25	25	18	18	25	0	0	0	0
Alarms	0	0	0	0	0	0	0	0	0	0	0	0
Wrong proofs	0	0	0	0	0	0	0	0	0	0	0	0
Wrong alarms	0	0	0	0	0	0	0	0	0	0	0	0
Timeouts	29	11	8	8	8	0	0	10	0	2	2	0
Out of memory	0	1	3	0	3	18	18	1	31	21	21	34
Other inconclusive	3	10	10	3	0	0	0	0	5	13	13	2

Table 4: Summary of the results for software verifiers on manually created tasks

did not see any change in its performance, as it solved 25 tasks with both default and ALAP configurations. CBMC did not solve any tasks with any configuration, for the same reason as in RQ1.

These results show that the ALAP-scheduling option is a useful technique for optimizing the translation of array write operations in BTOR2 models into C programs. By avoiding duplicate copies of arrays when possible, we reduce the memory consumption and verification time of the software analyzers. However, this option is not effective on the original benchmark set, due to a certain pattern of write operations that requires copies of arrays. Therefore, finding a way to handle this pattern is an important direction for future work.

RQ3: Complementing Effects for all Configurations

The third research question aims to investigate whether using different configurations of the Btor2-to-C translation flow can complement each other in solving more tasks by the verifiers. By complementing, we mean that a verifier can solve some tasks with one configuration that it cannot solve with another configuration, and vice versa. We compare the sum of the results for all configurations with the best result for each configuration. We use the same two sets of benchmarks and the same verifiers as in RQ1 and RQ2. We observe that some configurations can complement each other in solving more tasks by some verifiers, while others cannot. In particular, we note the following findings:

- On the original benchmark set, the balanced blasting, skewed blasting, and ALAP-scheduling configurations can complement each other in solving more tasks by AVR, Cbmc, and CPAchecker. AVR increases its number of solved tasks from 139 (the best result with balanced blasting) to 145 (the sum of all configurations). Cbmc increases its number of solved tasks from 19 (the best result with no blasting) to 20 (the sum of all configurations). Cbmc also finds a new alarm for the task 'picorv32_mutCY_mem-p8', which was previously unknown and confirmed to be true (the alarm) after further analysis. This alarm was found thanks to array blasting and ALAP-scheduling, which shows the value of these techniques for verification. CPAchecker increases its number of solved tasks from 6 (the best result with skewed blasting) to 7 (the sum of all configurations).
- On the manually created set, all configurations can complement each other in solving more tasks by AVR and CPAchecker. AVR increases its number of solved tasks from 32 (the best result with balanced blasting) to the maximum of 36. CPAchecker increases its number of solved tasks from 16 (the best result with skewed blasting) to 17. However, all these tasks are also solved by the ALAP-scheduling configuration, which achieves the best performance among all verifiers with 28 solved tasks.

These results show that using different configurations of the Btor2-to-C translation flow can complement each other in solving more tasks by some verifiers. However, this effect is not very significant and depends on the characteristics of the tasks and the verifiers.

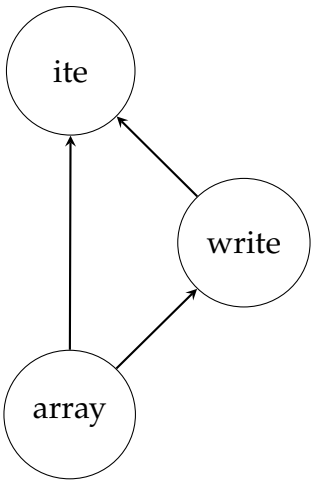

Figure 13: Common pattern makes copies of arrays necessary

6.5 Discussion

The previous section showed the experimental results of our evaluation of blasting and ALAP-scheduling techniques. We observed that these techniques have different effects on the performance of the verifiers, depending on the characteristics of the array tasks. We also noticed that the original benchmark set used by Btor2C did not allow us to fully exploit the potential of ALAP-scheduling, which motivated us to create a new set of manually crafted tasks. In this section, we will explain the reason behind this limitation and suggest a possible solution for future work in Chapter 7.

The main challenge that we faced when applying ALAP-scheduling to the original benchmark set was the presence of a specific pattern of write operations in most of the array tasks. This pattern consists of having both an array and a write operation to the same array in the same ite statement, as shown in Figure 13. This pattern makes it impossible for our algorithm to avoid creating duplicate copies of arrays, since the fan-out (ite) of the array that is being written into appears at a later index than the write operation in the schedule. Therefore, our algorithm always creates a copy of the array used for the write operation.

This pattern occurs in 174 out of 175 tasks in the original benchmark set, which means that ALAP-scheduling has no effect on these tasks. To overcome this problem, we created a new set of tasks that avoided this pattern and allowed us to demonstrate the benefits of ALAP-scheduling. However, this does not mean that ALAP-scheduling is useless for the original benchmark set. On the contrary, we believe that there is a way to handle this pattern and optimize the translation of array write operations. We will

discuss this idea in the following chapter, where we present our future work directions.

Additionally to the effectiveness of the blasting method which has successfully proven itself, using both options of blasting read operations into either skewed ite-tree-chains or balanced-ite-trees can increase the total number of solved tasks.

6.6 Threats to Validity

In this section, we discuss the potential threats to the validity of our evaluation and our techniques. We distinguish between external validity and internal validity. External validity refers to the extent to which our results can be generalized to other contexts and settings. Internal validity refers to the extent to which our results are free from errors and biases that may affect their reliability and accuracy.

6.6.1 External Validity

One of the threats to external validity is the selection of the benchmark sets that we used for our evaluation. We used two different sets of Btor2 tasks: the original benchmark set used by Btor2C, which contains 175 tasks from various sources such as hardware model checking competitions, and a manually created set of 36 tasks that we designed to test our ALAP-scheduling technique. We can assume that the original benchmark set is a representative sample of the relevant tasks in the verification community, since it was used in a previous work and covers a wide range of models and sources. However, the manually created set is not a realistic sample, since it was created with a specific purpose and does not reflect the complexity and diversity of real-world tasks. Therefore, we cannot claim that our results on the manually created set are conclusive or generalizable to other tasks. However, we still think that the manually created set is useful for illustrating the potential benefits of our ALAP-scheduling technique on tasks that can avoid duplications.

6.6.2 Internal Validity

One of the threats to internal validity is the correctness of our implementations. We implemented two techniques: array blasting, which is a Python script that transforms Btor2 tasks with arrays into arrayless equivalents, and ALAP-scheduling, which is an option in Btor2C that optimizes the translation of array write operations into C programs. We tested our implementations on both benchmark sets and compared the results. We did not observe any false proofs or alarms in our experiments, which suggests that our implementations are correct and consistent with Btor2C. However, we cannot guarantee that our implementations are free from errors or bugs. Moreover,

we did not have a large and diverse enough set of array tasks to test all possible scenarios and patterns that may occur in Btor2 models. Therefore, there may be some cases where our implementations may fail or produce incorrect results.

Another threat to internal validity is the unexpected behavior of Cbmc on a single task. We observed that Cbmc was able to find a correct alarm for a task that had an unknown verdict. When translating the task with the ALAP option enabled, no array duplications are avoided. This alarm was found only when we used ALAP-scheduling or array blasting on the task, but not when we used Cbmc with the default translated variant. This suggests that the ALAP technique somehow helped Cbmc to solve this task, even though no array copies are avoided. However, we could not explain why this happened or what was different between the translated programs that caused this behavior. We also could not reproduce this behavior on other tasks. Therefore, we cannot rule out the possibility that this behavior was caused by a faulty translation or a bug in the implemented algorithm.

7 Future Work

7.1 Array-Write-Ite Pattern Problem

One of the limitations of our evaluation was that the ALAP-scheduling technique did not have any effect on the original benchmark set used by Btor2C. This was because most of the array tasks in this set had a specific pattern of write operations that prevented our algorithm from avoiding copies of arrays. In this section, we will describe this pattern and propose a technique to handle it as a future work direction.

The pattern that we encountered in the original benchmark set was the following: an array is written into if a certain condition is satisfied, and otherwise the original array is returned. This pattern is represented by an if-then-else (ite) operation that takes two arrays as arguments: one that is a copy of the other with a single value overwritten at a given index, and the other that is the original array. An example of this pattern is shown in Figure 13.

Our algorithm for ALAP-scheduling creates a copy of the array used for the write operation, since the fan-out (ite) of the array that is being written into appears at a later index than the write operation in the schedule. Therefore, we end up with two arrays for this pattern: one for the write operation and one for the ite operation. This defeats the purpose of ALAP-scheduling, which is to reduce the number of arrays and avoid unnecessary copies.

To overcome this problem, we propose a technique that avoids creating a copy of the array and instead writes to the original array only when the condition that is being used in the ite operation is true. This way, we can reduce the number of arrays used for this pattern from two to one. The workaround for the C translation is shown in Figure 14. In lines 5 and 6, we present another way of translation for write operations that implements the workaround for this pattern.

To apply this technique, we need a reliable way of identifying these patterns in the Btor2 models and also an assurance that overwriting the original array does not result in information loss due to the overwritten value. This condition needs to be respected like it was in the implementation of ALAP-scheduling. The technique presented can potentially eliminate all array duplications for this pattern, which can improve the performance of the software analyzers.

We believe that this technique is a promising direction for future work, as it can

```
1  for (;;) {
2      // Getting external input values ...
3      // Creating intermediate signals ...
4      ...
5      SORT_1* write = old_array;
6      write[index] = ite_condition ? value : old_array[index];
7      // Assuming invariants ...
8      // Asserting properties ...
9      // Assigning next states ...
10 }
```

Figure 14: Avoiding copies for common ite pattern

significantly reduce array duplications for a large number of Btor2 tasks. We also think that this technique can be combined with array blasting to further enhance the verification of hardware models by software analyzers.

7.2 Avoiding Duplicates in the Initialization Process of States

As we explained in Section 3.3, Btor2 supports the `init` construct, which allows us to specify the initial values of states in a model. When Btor2C translates a Btor2 model into a C program, it performs the initialization process before the sequential circuit part, which is simulated by an infinite loop, as shown in Figure 14. However, our ALAP-Scheduling algorithm does not apply to the write operations that occur in the initialization process. Therefore, the translated C program may still create duplicate copies of arrays during the initialization process, which can affect the performance of the software analyzers.

To improve the translation of array write operations in the initialization process, we could extend our ALAP-Scheduling algorithm to this part of the code as well. This would require us to analyze the circuit models and ensure that the write operations that appear in the `init` part do not interfere with the write operations that appear in the sequential circuit part. Since we want to preserve the correctness and completeness of the translation, we need to be careful about this extension and test it thoroughly on different models.

We believe that this extension is a worthwhile direction for future work, as it can further reduce the number of array duplications in the translated C programs and enhance the verification of hardware models by software analyzers.

7.3 Using Integer Linear Programming for Finding the Optimal Schedule

One of the limitations of our algorithm for ALAP-scheduling is that it does not guarantee the creation of an optimal schedule. An optimal schedule is one that minimizes the number of array duplications for a given model. Our algorithm may create suboptimal schedules when there are multiple write operations on the write stack, which is a data structure that stores the write operations that have not been scheduled yet. In this section, we will illustrate this problem and propose a technique to solve it as a future work direction. The problem arises when there are two or more write operations on the write stack that write into different arrays. A simplified example of this situation is shown in Figure 15. In this example, there are two write operations on the write stack: write_1 and write_2. write_1 writes into array_1 which has read_2 as its fan-out. write_2 writes into array_2 and has read_1 as its fan-out which is an argument node of read_2. Our algorithm has to decide which write operation to schedule first, since this may affect the number of array duplications. If it schedules write_2 first, then it can also schedule read_2, which is the fan-out of array_1 before write_1. This way, it avoids creating a duplicate of of array_1 for write_1, since no access operation to array_1 can be done after write_1. However, if it schedules write_1 first, then it cannot schedule read_2 yet, since it depends on write_2. This means that read_2 will be scheduled later than write_1, which requires creating a duplicate of array_1, since array_1 will be accessed after write_1. This is a complicated problem that can be solved by using integer linear programming (ILP). ILP is a technique that can find an optimal solution for a problem that involves integer variables and linear constraints and objectives. In our case, we can formulate the problem as follows: each write operation has a binary variable α that indicates whether a duplication is required for its old array or not. The objective is to minimize the sum of all α variables over all write operations. The constraints are based on the dependencies between the operations and the order of the schedule. By using ILP, we can find an optimal schedule that minimizes the number of array duplications for any model. This would improve our algorithm for ALAP-scheduling and enhance the verification of hardware models by software analyzers. However, ILP is not a trivial technique to apply and may require additional tools and libraries to implement and solve. Therefore, we leave this as a direction for future work.

7.4 Identifying the Root Cause of the Array Problem

During this thesis work, research was done on identifying the root cause for analyzers reaching their timelimits when used on array tasks. To this end, we conducted some

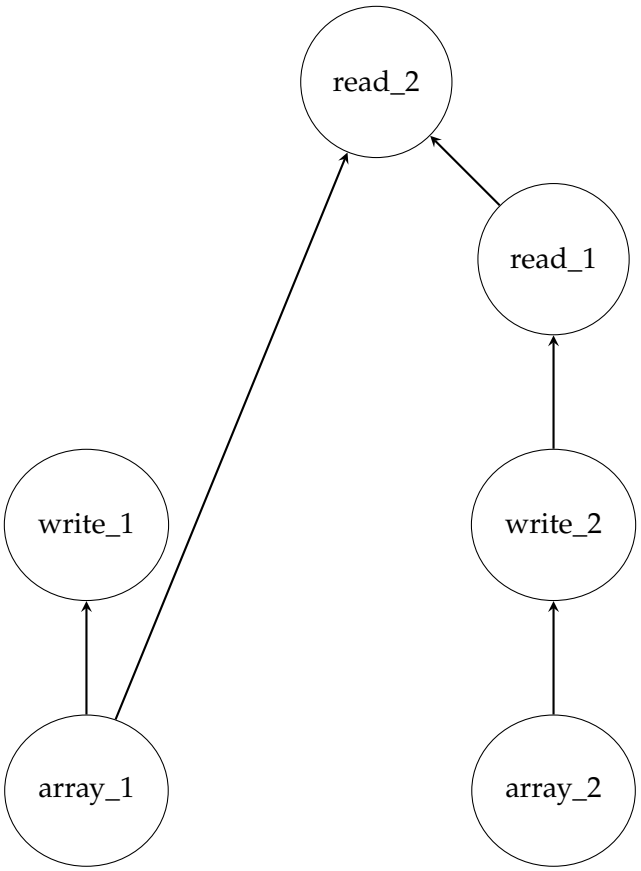

Figure 15: With both writes on the stack, popping write_2 first helps avoiding a duplicate (simplified)

experiments with CPAchecker. We used one of the C programs that we translated from the manually created task 'talk_index_size_10' as a test case. We found that CPAchecker was unable to solve this task within 300 seconds, and that most of the time (290 seconds) was spent on a single SMT query that was sent to the SMT solver MathSAT5 [3], which is used by CPAchecker. However, when we tried the same query with another SMT solver, cvc5 [8], it was solved in less than a second. This showed a significant performance gap between these two SMT solvers for this particular query. These runs were done on different machines with varying PC specifications.

We suspect that this performance gap may be due to some differences in how these solvers handle certain theories or features that are relevant for array tasks. However, we did not have enough time to investigate this issue in depth and compare these solvers on other array tasks. Therefore, we suggest that as a future work direction, a more systematic and comprehensive comparison of different SMT solvers on array tasks should be done, using the same experimental setup and environment. This could help us understand the root cause of the array problem better and find ways to overcome it.

8 Conclusion

The main goal of this thesis was to improve the verification of hardware models by software analyzers. To achieve this goal, we proposed and implemented two techniques that preprocess the hardware models in BTOR2 format and transform them into more suitable formats for verification. The first technique is array blasting, which eliminates arrays from the models and replaces them with bit-vector operations. The second technique is ALAP-scheduling, which optimizes the translation of write operations into C programs. We evaluated our techniques on a set of more than 300 array tasks, using different configurations with state-of-the-art verifiers. We also created a new set of manually crafted tasks to test our techniques more thoroughly. Our evaluation showed that our techniques have significant benefits for the verification of hardware models by software analyzers. In particular, we found that:

- Array blasting enables the translation of BTOR2 array tasks into AIGER files (with BTOR2AIGER), which was not possible before. This allows us to use bit-level hardware verifiers such as ABC on these tasks, which can find more alarms and proofs than software verifiers.
- Array blasting also increases the number of proofs for software verifiers from zero to seven on the original benchmark set. This shows that array blasting can make the tasks more amenable to verification by software analyzers.
- Different configurations can complement each other in solving more tasks by some verifiers. For example, AVR increases its number of solved tasks from 139 to 145 on the original benchmark set, and from 6 to 7 in the case of CPACHECKER, by using both balanced and skewed blasting options.
- ALAP-scheduling does not have any effect on the original benchmark set, due to a specific pattern of write operations that prevents our algorithm from avoiding copies of arrays. However, on the manually created set, which avoids this pattern, ALAP-scheduling achieves better results than blasting among all verifiers with 25 solved tasks.

These results demonstrate that our techniques are valuable contributions to the field of formal verification and enhance the verification of hardware models by software

analyzers. However, our techniques are not perfect and have some limitations and challenges. Therefore, we also suggested some directions for future work, such as finding a way to handle the write-ite pattern problem, extending the ALAP-scheduling algorithm to the initialization process of states, and using integer linear programming for finding the optimal schedule.

We hope that this work will inspire further research on this topic and lead to a solution that solves the array problem for all analyzers and makes our techniques obsolete.

Data-Availability

For transparancy purposes, all results reported in this paper together with the analyzers and their respective versions used, as well as our implementations are available in the reproduction package. To reproduce the results, clone and follow the instructions of following repository: https://gitlab.com/btor2c-array-encoding/blasted-hwmc-evaluation. Direct interaction with the experimental results shown in Tables 1 and 2 can be done by visiting https://www.cip.ifi.lmu.de/~atess/experimental-results/paper-comparison-results/tab1.all-verifiers.table.html#/ and for the results of the manually created tasks by visiting https://www.cip.ifi.lmu.de/~atess/experimental-results/manually-created-tasks-results/tab1.all-verifiers.table.html#/.

Abbreviations

ALAP as-late-as-possible
bv bit-vector

List of Figures

List of Tables

Bibliography

[1] A. Biere. *The AIGER And-Inverter Graph (AIG) format version 20071012*. Tech. rep. 07/1. Institute for Formal Models and Verification, Johannes Kepler University, 2007. DOI: 10.35011/fmvtr.2007-1.

[2] A. Biere, N. Froleyks, and M. Preiner. *11th Hardware Model Checking Competition (HWMCC 2020)*. http://fmv.jku.at/hwmcc20/index.html. (Visited on 09/14/2023).

[3] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. "The MathSAT5 SMT Solver." In: *Proc. TACAS*. LNCS 7795. Springer, 2013, pp. 93–107. DOI: 10.1007/978-3-642-36742-7_7.

[4] A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. "Software Verification Using k-Induction." In: *Proc. SAS*. LNCS 6887. Springer, 2011, pp. 351–368. DOI: 10.1007/978-3-642-23702-7_26.

[5] A. Goel and K. Sakallah. "AVR: Abstractly Verifying Reachability." In: *Proc. TACAS*. LNCS 12078. Springer, 2020, pp. 413–422. DOI: 10.1007/978-3-030-45190-5_23.

[6] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. "Btor2, BtorMC, and Boolector 3.0." In: *Proc. CAV*. LNCS 10981. Springer, 2018, pp. 587–595. DOI: 10.1007/978-3-319-96145-3_32.

[7] M. Afzal, A. Asia, A. Chauhan, B. Chimdyalwar, P. Darke, A. Datar, S. Kumar, and R. Venkatesh. "VeriAbs : Verification by Abstraction and Test Generation." In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1138–1141. DOI: 10.1109/ASE.2019.00121.

[8] H. Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by D. Fisman and G. Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24.

[9] D. Beyer. "Progress on Software Verification: SV-COMP 2022." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by D. Fisman and G. Rosu. Cham: Springer International Publishing, 2022, pp. 375–402. ISBN: 978-3-030-99527-0. DOI: 10.1007/978-3-030-99527-0_20.

[10] D. Beyer, P.-C. Chien, and N.-Z. Lee. "Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by S. Sankaranarayanan and N. Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 152–172. ISBN: 978-3-031-30820-8. DOI: 10.1007/978-3-031-30820-8_12.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs." In: *Proc. TACAS*. LNCS 1579. Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14.

[12] A. Biere, T. van Dijk, and K. Heljanko. "Hardware model checking competition 2017." In: *2017 Formal Methods in Computer Aided Design (FMCAD)*. 2017, pp. 9–9. DOI: 10.23919/FMCAD.2017.8102233.

[13] C. Wolf. *Yosys Open SYnthesis Suite*. https://yosyshq.net/yosys/. (Visited on 09/14/2023).

[14] D. Beyer and M. E. Keremoglu. "CPAchecker: A Tool for Configurable Software Verification." In: *Proc. CAV*. LNCS 6806. Springer, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16. https://www.sosy-lab.org/research/pub/2011-CAV.CPAchecker_A_Tool_for_Configurable_Software_Verification.pdf.

[15] D. Beyer, S. Löwe, and P. Wendler. "Reliable Benchmarking: Requirements and Solutions." In: *Int. J. Softw. Tools Technol. Transfer* 21.1 (2019), pp. 1–29. DOI: 10.1007/s10009-017-0469-y. https://www.sosy-lab.org/research/pub/2019-STTT.Reliable_Benchmarking_Requirements_and_Solutions.pdf.

[16] E. M. Clarke, D. Kröning, and F. Lerda. "A Tool for Checking ANSI-C Programs." In: *Proc. TACAS*. LNCS 2988. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided abstraction refinement for symbolic model checking." In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[18] N. Eén, A. Mishchenko, and R. K. Brayton. "Efficient implementation of property directed reachability." In: *Proc. FMCAD*. http://dl.acm.org/citation.cfm?id=2157675. FMCAD Inc., 2011, pp. 125–134.

[19] E. Giunchiglia, P. Marin, and M. Narizzano. "sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning." In: *Theory and Applications of Satisfiability Testing – SAT 2010*. Ed. by O. Strichman and S. Szeider. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 85–98. ISBN: 978-3-642-14186-7. DOI: 10.1007/978-3-642-14186-7_9.

[20] *IEEE Standard for Verilog Hardware Description Language*. 2006, pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[21] ISO/IEC JTC 1/SC 22. *ISO/IEC 9899-2018: Information technology — Programming Languages — C*. https://www.iso.org/standard/74528.html. International Organization for Standardization, 2018.

[22] K. L. McMillan. "Lazy Abstraction with Interpolants." In: *Proc. CAV*. LNCS 4144. Springer, 2006, pp. 123–136. DOI: 10.1007/11817963_14.

[23] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. "ESBMC 5.0: An Industrial-Strength C Model Checker." In: *Proc. ASE*. ACM, 2018, pp. 888–891. DOI: 10.1145/3238147.3240481.

[24] R. Brayton and A. Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool." In: *Proc. CAV*. LNCS 6174. Springer, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[25] R. Jhala and R. Majumdar. "Software Model Checking." In: *ACM Computing Surveys* 41.4 (2009). DOI: 10.1145/1592434.1592438.

[26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy abstraction." In: *Proc. POPL*. ACM, 2002, pp. 58–70. DOI: 10.1145/503272.503279.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. "Abstractions from proofs." In: *Proc. POPL*. ACM, 2004, pp. 232–244. DOI: 10.1145/964001.964021.

[28] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. "Automatic Predicate Abstraction of C Programs." In: *Proc. PLDI*. ACM, 2001, pp. 203–213. DOI: 10.1145/378795.378846.