Bitvector Support in the Theta Formal Verification Framework

Mihály Dobos-Kovács*, Ákos Hajdu*, András Vörös*
*Department of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
vori@mit.bme.hu

***Abstract*—The verification of safety-critical software systems has many challenges, such as the complex language constructs in embedded software. This paper addresses the verification problem of software systems using bitwise operations, and we present an extension to the Theta open-source formal verification framework. Our goal is to integrate bitvectors and bitwise operations in the abstraction-refinement-based formal verification methods in Theta. Supporting bitvectors is a step towards the verification of industrial embedded software systems. We extended the language support in Theta with formal semantics. In addition, the new language constructs and operators are transformed into the formal language inside Theta. We also need new algorithms to solve the verification problem: we implemented Newton-style refinement algorithms in Theta to verify software with bitvectors and bitwise operators efficiently.**

***Index Terms*—formal verification, bitvector, Newton refinement, Theta**

I. Introduction

Nowadays, software systems are becoming more and more part of our everyday lives. One distinguished category is safety-critical systems, whose correct functioning must be ensured, as an error can have huge financial costs or damage human lives. To ensure the safety of critical systems, formal verification methods are often used to find errors.

Theta [1] is a formal verification framework developed at the Budapest University of Technology and Economics. It follows a modular approach and provides abstraction-refinement-based algorithms for the reachability analysis of multiple formalisms. Moreover, Theta has multiple language frontends that enable the formal verification of C code or statechart models.

Theta has powerful abstraction-based verification algorithms, where solvers are used to refine the abstraction to produce proof of correctness or find an error trace. Solver technology is an essential part of the verification approach, and many challenges have to be solved.

Theta uses *Satisfiability Modulo Theory (SMT)* [2] solvers: one such solver is Z3 [3], developed by Microsoft. These SMT solvers receive an SMT problem as input and determine whether a variable assignment satisfies the constraints. For a satisfiable variable assignment, the solvers produce a model that satisfies the constraints in the problem. In case the problem is unsatisfiable, the solver can calculate an unsatisfiable core or an interpolant. It is important to note that not all solvers support all kinds of SMT input or even all operations on their supported inputs.

Theta aims at the verification in various domains. However, dealing with problems originating from industrial partners revealed that handling some language constructs is not satisfactory: verification attempts failed or yielded incorrect results due to the unsupported constructs and operations.

An integral part of this issue was the way integers are modeled in SMT solvers. In a computer, an integer is stored on a finite amount of bits. As a result, the range of integers in a computer is finite, and integers even tend to overflow or underflow at the boundaries of their range. Moreover, computers allow bit-level access to integers, enabling such integers to represent bitfields. On the other hand, integers in SMT problems are based on the mathematical set of integers. An SMT integer has an infinite range, so overflow and underflow can not happen. It is also not supported to access the individual bits in mathematical integers (however, with some tricks, one can solve bit-level access).

To bridge this gap in semantics, we introduced bitvector support in Theta. Bitvectors are capable of capturing the semantics of computer integers in SMT problems.

An example for the case when overflow matters can be seen in Listing. 1. If the variable x is treated as a mathematical integer, then $255 + 1 \neq 0$. However, if we consider that the unsigned char type in C stores the variables on 8 bits and the variable overflows, then the assertion will hold (mathematically $255 + 1 \, mod \, 2^8 = 0$).

Listing 1. Example for a program code where overflow matters

```
void main() {
  unsigned char x = 255;
  x = x + 1;
  assert(x == 0);
}
```

Supported by the ÚNKP-21-2 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

This research was funded by the European Commission and the Hungarian Authorities (NKFIH) through the Arrowhead Tools project (EU grant agreement No. 826452, NKFIH grant 2019-2.1.3-NEMZ ECSEL-2019-00003).

II. Background

A. SMT

Satisfiability Modulo Theory (SMT) [2] problems are based on first-order logic. Although a first-order formula is generally undecidable algorithmically, specific theories [4] give interpretation to the symbols in a first-order formula. These theories loosen the underlying problem, making the first-order satisfiability problem decidable.

An SMT problem is a decision problem for first-order formulas. Given a formula and a set of theories, an SMT solver can decide whether there exists a substitution of symbols to concrete values in the formula, so after the substitution, the formula evaluates to true. If such substitution exists, the problem is satisfiable, and the substitution is called a model of the problem. If no such substitution exists, the problem is unsatisfiable, and the unsatisfiable core or an interpolant can be calculated for the problem.

There are various SMT theories [4]. In the following list, we present the most notable ones:

- Core: Defines the boolean type and operations. A boolean symbol can have true or false as a value.
- Ints: Defines the integer type based on the mathematical set of integers and operations.
- Reals: Defines the real type based on the mathematical set of rationals and operations.
- ArraysEx: Defines functional arrays with extensibility and operations on them.
- FixedSizeBitvectors: Defines bitvectors with arbitrary size and operations on them.

B. Integers vs. Bitvectors

An SMT integer value is based on the mathematical set of integers. It means that for an integer $v \in \mathbb{Z}$ holds. Multiple operations are defined on it, all based on the corresponding mathematical operation: addition, subtraction, multiplication, (integer) division, modulo, and comparison.

On the other hand, a bitvector can be thought of as a bitfield. Each bitvector has a positive fixed size that describes how many bits it consists of. The value of each bit is 0 or 1 (or false or true).

A bitvector - as a set of bits - has three (two and a half) different interpretations:

- It is a collection of independent bits. This interpretation supports operations like concatenating bitvectors, extracting one or more bits, and the traditional bitwise operations as well: bitwise and, or and xor.
- It is the binary form of an unsigned number. This unsigned number overflows (so $111...111_2$ is followed by $000...000_2$). This interpretation has the traditional arithmetic and relational operators defined: the same as the integers.
- It is the binary form of a signed number in two's complement form. This signed number overflows (so $111...111_2$ is followed by $000...000_2$). This interpretation has the traditional arithmetic and relational operators defined: the same as the integers.

TABLE I
Bitvector Operations

1100_2	$+$	1110_2	$=$		
-4	$+$	-2	$=$	1010_2 (-6)	(signed) addition
12	$+$	14	$=$	1010_2 (10)	(unsigned) addition
1010_2	$/$	0010_2	$=$		
-6	$/$	2	$=$	1101_2 (-3)	signed division
10	$/$	2	$=$	0101_2 (5)	unsigned division
1010_2	$\&$	0010_2	$=$	0010_2	bit-wise and

It can be seen that the latter two options are almost identical. Due to two's complement, some arithmetic operations are the same for the unsigned and signed variant (e.g., addition, multiplication). At the same time, some operations require different semantics for the unsigned and signed cases (e.g., division or relational operations).

An example for bitvector operations can be seen in Table I. The first example adds two numbers. As it can be seen, the binary representation of the result is the same regardless of whether the operations were signed or unsigned. However, there is a difference in the binary representation in the second example. The division operation differs in signed and unsigned cases. The third example shows the bitwise and operation, where numeric representation does not matter, as the operation is performed directly on the bits.

C. Model checking

Model checking [5] [6] is a formal method that decides whether a formal requirement holds on a formal model. The model is correct if a mathematical proof exists that the requirement holds on the model. It is incorrect if mathematical proof exists that the requirement does not hold. It is worth to be noted that the proof of incorrectness is often a (counter)example.

Computer programs are often formalised as Control Flow Automaton (CFA) [7]. A CFA is a (V, L, l_0, E) tuple, where:

- $V = \{v_1, v_2, ...\}$ is the set of variables present in the program. Each v_i variable has a D_{v_i} domain,
- $L = \{l_0, l_1, ...\}$ is the set of control locations. They can be interpreted as the possible values of the program counter,
- $l_0 \in L$ is the initial location, where the program starts,
- $E \subseteq L \times Ops \times L$ is the set of transitions, where L is the set of control locations, and Ops is the set of operations. A transition is a directed edge between two locations, with an operation labeling it. The operation can be:
  - A $v := expr$ deterministic assignment, where the value of variable v will be set to the value of the evaluated expression $expr$,
  - A $havoc\, v$ non-deterministic assignment, where the value of variable v will be set to an arbitrary value taken from its domain. Non-deterministic assign-

ments are helpful when modeling data coming from the user.
  - A $[cond]$ guard; a transition with a guard can only be executed if the expression $cond$ evaluates to true.

To give a formal requirement, some of the locations of the CFA tend to be labeled as an error location. In this case, the formal requirement is that execution never reaches an error location. An example for a CFA can be seen in Fig. 1. As it can be seen, the error location l_e was derived from the assertion in Listing 1.

Considerably simplified, model checking algorithms check all reachable states of the system under verification and check whether the states associated with requirement failure (error location) are reachable. A (concrete) state of a CFA is a $(l_i, d_1, ..., d_n)$ tuple, where:

- $l_i \in L$ is the current location,
- $d_1, d_2, ..., d_n$ are the current values of the variables, where $d_i \in D_{v_i}, n = |V|, v_i = d_i$.

However, even the simplest systems can have a huge or even infinite state space. This phenomenon is called state-space explosion, and it significantly hinders model checking methods.

D. CEGAR

To combat the state-space explosion problem, many model checking methods use some kind of abstraction to simplify the state-space representation. These algorithms work on the abstract state-space that consists of abstract states. An abstract state represents a set of concrete states.

There are multiple methods to associate concrete states with abstract states. The abstract domain describes this method. Each abstract domain effectively addresses different kinds of problems, and choosing the correct one is a complex problem. Some abstract domains can be:

- Predicate abstraction [8] is a common abstract domain that takes first-order formulas as predicates and groups concrete states by predicate values into a single abstract state.
- Explicit value analysis [9] is another common abstract domain that follows the concrete values of some chosen variables instead of predicates.

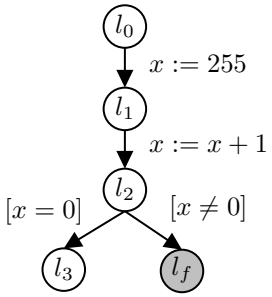

Fig. 1. The code from Listing 1. in CFA format

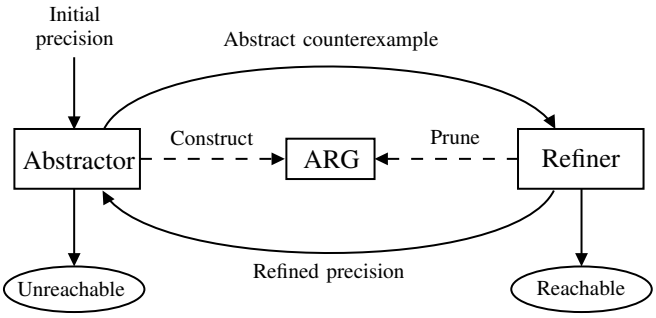

Fig. 2. The CEGAR loop

- Product abstractions [10] tend to combine the previous two methods.

The core of the analysis in the Theta framework is the model checking method *Counterexample-guided Abstraction Refinement (or CEGAR for short)* [11]. CEGAR performs reachability analysis over a given abstract domain. In the core of the algorithm stands its central data structure, the so-called *Abstract Reachability Graph (ARG)*.

The ARG is constructed according to the operations in the CFA. The ARG consists of nodes and two kinds of edges:

- the nodes are annotated with abstract states, that are overapproximations of concrete, possibly reachable states on a given path,
- the forward edges are annotated with actions (transitions), that are interpreted on the states,
- the backward edges, also called covering edges, represent that we have already traversed that part of the state space.

Two components manipulate the ARG in a loop (as seen in Fig. 2.): the abstractor and the refiner. At the start of CEGAR, the ARG is created with an initial node, and an initial precision is given, which is used by the abstraction process. The definition of precision depends on the abstract domain: in the case of predicate abstraction, it can be a set of predicates; in the case of explicit value analysis, the set of variables to follow.

In each iteration, the abstractor constructs the ARG by either expanding nodes or covering them. The former uses forward edges and builds a path step-by-step from the initial node. At the same time, the latter describes that the node is covered by another, through which all states are reachable that are reachable from the covered node for the given precision.

If the abstraction process terminates and does not reach any (abstract) error states (i.e., states modeling requirement failure), then the built ARG serves as evidence for the unreachability of the unsafe states, so the system is safe. However, if an unsafe (abstract) state is reached, then it is analyzed by the refiner to check if it is reachable through a concrete, executable trace of the software.

The refiner first concretizes the abstract path (or abstract counterexample) from the initial node to the given unsafe node. If the path is feasible, then the concrete path, also called

counterexample, is proof of the reachability of the unsafe state, so the system is unsafe. On the other hand, should the abstract path be infeasible (with other words spurious), the abstraction needs to be refined because the overapproximation of the abstraction denotes some abstract states to be reachable while some of their underlying concrete states are unreachable. This refinement can be achieved by pruning the ARG, uncovering covered nodes, or computing a new precision.

III. Architecture and Implementation

A. Architecture of Theta

Theta can receive problems in various engineering formats, including C or statecharts. It converts the input to one of its supported formal representations using a dedicated frontend. Theta supports the formal representations *Control Flow Automata (CFA)* [7], *Symbolic Transition Systems (STS)* [12], or *Timed Automata (XTA)* [13].

The core of Theta is the abstraction-refinement-based CEGAR algorithm. For the sake of extensibility, the CEGAR algorithm is independent of the formalisms. However, it still needs information from the formalisms. This information is provided to the CEGAR algorithm by interpreters (see Fig. 3), which depend on a specific formalism. The interpreter provides:

- The *init function*, which returns the abstract initial states. It applies the initial precision to the initial control location of the CFA.
- The *transfer function* calculates the abstract successor states of a state given an action. In the case of a CFA, the transfer function uses the location and transition associated with the state and action to calculate the successor states.
- The *action function* that returns the set of enabled actions from a given abstract state. The interpreter of the CFA uses the guards to determine the set of enabled transitions.

The core of many algorithms in Theta are SMT problems. The transfer functions typically use queries to the SMT solver to compute the successor states. The refiner also uses the SMT solver to check the feasibility of the abstract counterexample, and interpolation or unsatisfiable cores are used to refine the precision.

Theta provides a generic SMT solver interface that the analysis algorithms can depend on. This interface supports incremental solving, unsatisfiable cores, and interpolants as well. Typically, the interpreter calls the SMT solver interface when abstract successor states need to be calculated or by the refiner when the feasibility of an abstract path needs to be checked. Theta currently provides only one implementation of the SMT solver interface, which uses an older version of Microsoft's Z3 [3] solver.

Expressions are another important aspect of Theta. The expressions are similar to the expressions found in any programming language: there are literals, unary, binary, ternary, and other kinds of expressions. Each expression has a type (that might depend on its operands) and defines specific

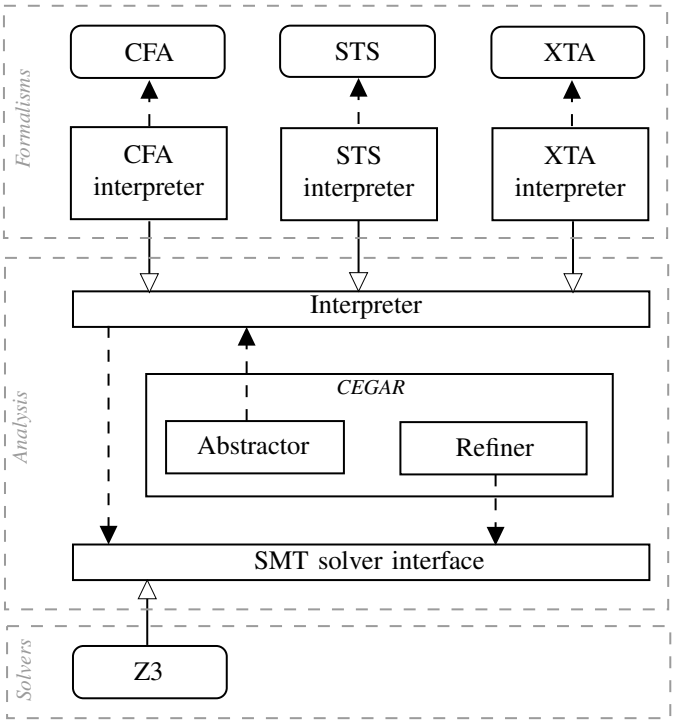

Fig. 3. The architecture of Theta

semantics. For example, there is a different expression for the addition of integers and rationals as they are semantically different operations.

The language frontend converts the expressions of the programming language to the expressions in the formal representation inside Theta. The algorithms heavily use theta expressions. To invoke a query on the SMT solver, the Theta expressions need to be transformed. The expression will be unfolded, meaning each variable is replaced by a symbol that may be assigned only once (similarly to static single assignment). The result of SMT queries is then folded back (the inverse operation of unfolding) by the algorithms and acted upon. An example of an expression in different roles can be seen in Fig. 4.

B. Implementing Bitvectors

Expressions are a core element of Theta. They are part of the formalisms used in the abstractor and refiner and passed to the SMT solver. To provide support for bitvectors, a new Theta type and new kinds of Theta expressions are introduced.

When considering a bitvector, two parameters need to be considered. The first is the *length* of the bitvector in bits, while the second is the *signedness* of the bitvector. As mentioned earlier, bitvectors mimic the integer representation of computer languages, and computer languages use two's complement representation to store signed values.

Due to the two's complement representation, there are operations, like addition, subtraction, multiplication, etc., that act the same, should the value be signed or unsigned. However,

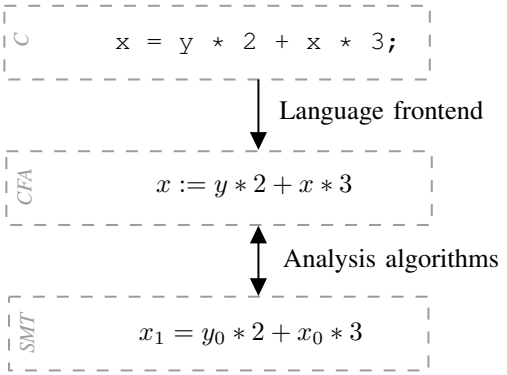

Fig. 4. The role of expressions in Theta

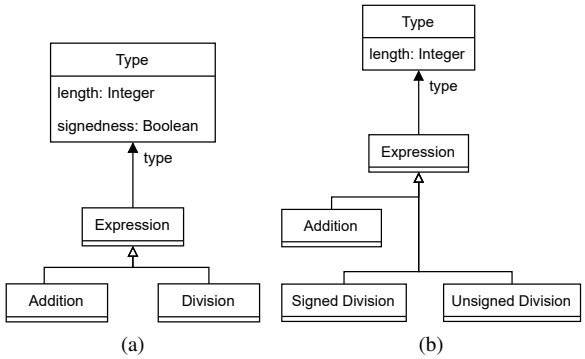

Fig. 5. Two ways of representing signedness

operations, typically comparison operators or divisions, behave differently based on the signedness of the value. See an example in Table I.

The information about signedness can be stored in two locations (see in Fig. 5):

- (a) in the type of the expression, and each multi-operand expression expects its operand to match their signedness,
- (b) or in the expression kinds themselves, so if an operation behaves differently for signed and unsigned values, it will become two distinct kinds of expression.

The advantage of the first solution is that it prohibits an accidental operation between a signed and an unsigned value, and it mimics the semantics of the C-like languages. On the other hand, the benefit of the second solution would be low overhead, as the SMT theory FixedSizeBitvectors uses the same semantics.

We decided to implement the second solution mainly for the reason of SMT compatibility. Thus, the new bitvector type has only a single parameter that describes its length, and the signedness information is encoded in the expressions. Multiple new kinds of expressions were also introduced, such as arithmetic operations (addition, subtraction, multiplication, signed division, unsigned division, etc.), bitwise operations (bitwise and, bitwise or, bit extraction, extension, etc.), and comparison operations (signed greater-than, unsigned greater-than, etc.).

After introducing new expressions, other parts of Theta needed to be expanded. Fortunately, the changes are limited to serialization and SMT solver implementations. The former means that the grammars need to be extended to handle bitvector expressions, while the latter means that each new expression kind needs to be mapped to an SMT operation.

IV. Newton Refinement

Currently, only one SMT solver is integrated into Theta, and that is Microsoft's Z3 [3]. To be more precise, Z3 version 4.5.0 is integrated with Theta. However, a significant issue with this version is that it does not support interpolation with bitvectors. Z3 has developed significantly over the past couple of years. Unfortunately, Z3 deprecated its interpolation support and completely removed it by 4.8.0 (October of 2018). It has never supported interpolation with bitvectors, and it is not expected to do so in the foreseeable future.

CEGAR depends on a refinement strategy, and as it happens, interpolation-based methods [14] are an effective option for refinement. Theta supports forward binary, backward binary, and sequence interpolation, among other solutions when it comes to refinement, so the fact that interpolation support is missing for bitvectors is a significant issue.

This problem can be addressed in two different ways: either by integrating different solvers that support interpolation with bitvectors into Theta or by implementing new refinement strategies that do not depend on interpolation. In this paper, we will discuss the second solution in detail. An issue with the first approach is that solvers tend to be specialized in certain theories, and a solver strong in bitvector category might not be adequate for other problems. Moreover, an advantage of this choice is that these new interpolation-free refinement strategies could be used later with solvers that do not have interpolation support at all. A particular solver in mind is CVC4 [15], which performed best in the bitvector category at *The International Satisfiability Modulo Theories Competition 2020 (SMT-COMP)* [16], or the newer versions of Z3.

We introduced the Newton refinement strategies [17] into Theta as they are not based on interpolation. Instead, they use weakest preconditions [18] or strongest postconditions [19] and unsatisfiable cores [4] for refinement.

A. Refinement Architecture

The refiner receives an abstract counterexample from the abstractor, and the task of the refiner is to concretize, i.e., execute the counterexample. Should the abstract counterexample be feasible (i.e., a concrete counterexample), then an error location is reachable, and the algorithm terminates with an unsafe result. Otherwise, the ARG is pruned, and the precision of the abstractor is refined.

Under the hood (see in Fig. 6.), it is the responsibility of the *Expression trace checker* to concretize the counterexample and check its feasibility. If the counterexample is feasible, the expression trace checker extracts the valuations of the concrete counterexample and creates a concrete trace to prove the

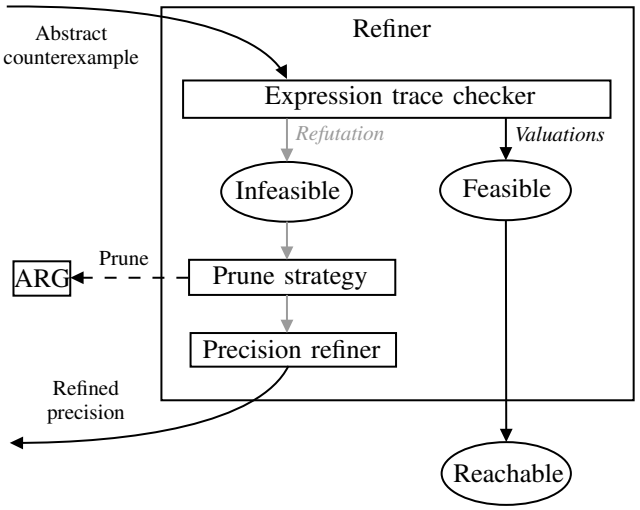

Fig. 6. The architecture of the refiner in Theta

reachability of the error location. However, if it is infeasible, it uses the refutation of the feasibility first to prune the ARG using a configurable *Prune strategy*, then refine the precision using an also configurable *Precision refiner*.

The expression trace checker to use can be selected out of a set of algorithms. Options include but are not limited to backward and forward binary or sequence interpolation; these options emit a list of predicates for the trace as refutation. On the other hand, there is currently an unsatisfiable core-based method that returns a list of variables for every trace step as a refutation; thus, it can only be used for explicit value analysis [9], not predicate abstraction [8].

Pruning happens if the abstract counterexample is infeasible, so there are states in the ARG that are not reachable. The goal of pruning is to remove all unreachable states from the ARG. A trivial solution is to remove everything from the ARG and start constructing it again; however, this solution is not the most effective. There are other refutation-dependent solutions as well.

Finally, the precision refiner refines the precision by using information from the refutation. The precision refiner to use is determined by the kind of abstraction and expression trace checker (so kind of refutation) used.

B. Introducing Newton Refinement

Newton style refinement in abstraction-refinement-based methods historically predated interpolation-based methods. Back in the day, interpolation-based methods were more effective, so they became widespread. However, in a relatively new publication, Dietsch et al. (2017) [17] showed that there are circumstances in which Newton refinement is competitive.

The input of the Newton algorithm is an infeasible concrete trace, which is an alternating list of locations and transitions starting with the initial location and ending with one of the error locations. First, the trace formula is constructed from the trace. The trace formula is $F = F_0 \wedge F_1 \wedge ... \wedge F_n$, where F_i is computed from the i$^{\text{th}}$ transition of the trace:

$$F_i = \begin{cases} \text{unfold}(cond), & \text{if transition}_\text{i} \text{ is } [cond] \\ x_{k+1} = \text{unfold}(e), & \text{if transition}_\text{i} \text{ is } x := e \\ true, & \text{if transition}_\text{i} \text{ is } havoc\ x \end{cases}$$

The function unfolds the expression by substituting its variables for indexed constants. Variable x is replaced by indexed constant x_k, where index k of variable x is the number of (deterministic and non-deterministic) assignments on the trace before the actual transition.

As the trace is infeasible, the trace formula is unsatisfiable [17]. The trace formula is a conjunction of other formulas, so the unsatisfiable core will be a subset of those formulas: $UC \subseteq \{F_0, F_1, ..., F_n\}$. The following subsections describe the refinement steps.

1) Step IT: The infeasible trace might be abstracted using the unsatisfiable core computed earlier. The trace is abstracted step-by-step:

$$\text{abstract transition}_\text{i} = \begin{cases} \text{transition}_\text{i}, & \text{if } F_i \in UC \\ [true], & \text{if } F_i \notin UC \text{ and} \\ & \text{transition}_\text{i} \text{ is } [cond] \\ havoc\ x, & \text{if } F_i \notin UC \text{ and} \\ & \text{transition}_\text{i} \text{ is } x := e \\ havoc\ x, & \text{if } F_i \notin UC \text{ and} \\ & \text{transition}_\text{i} \text{ is } havoc\ x \end{cases}$$

2) Step SP-WP: Next, an inductive sequence of assertions is constructed using either the original or the abstracted trace. The inductive sequence of assertions is created via the strongest postcondition or the weakest precondition predicate transformer operators. The result of the strongest postcondition post(transition, P) is another predicate P', which will always be true if the transition is applied to P. The result of the weakest precondition operator is analogous, pre(transition, P) is a predicate P' so that P will always be true if the transition is applied to P'. The strongest postcondition is applied from the start of the trace starting with the predicate $true$, while the weakest precondition is from the end of the trace starting with the predicate $false$ to create an inductive sequence of assertions from the predicates.

3) Step LV: The final, optional step is to project the inductive sequence of assertions created from the predicates to live variables. A variable is future-live at transition t_i in the trace if there is a future transition that reads the variable, and there is no future transition that writes it. A variable is past-live at transition t_i in the trace, if there is a past transition that reads or deterministically writes the variable, and there is no past transition that non-deterministically writes it. After computing, the set of future- or past-live variables for all transitions in the trace, each predicate in the inductive sequence of assertions is projected to the set of live variables using existential projection

for the future-live variables and universal projection for the past-live variables.

Combining the steps from before, the algorithm of the Newton refinement is the following:

1) Compute the trace formula of the infeasible trace
2) *IT* Optionally, calculate the unsatisfiable core for the trace formula and abstract the trace. Use the abstract trace in later steps.
3) Choose one of the following:
   a) *SP* Compute the inductive sequence of assertions using the strongest postcondition.
   b) *WP* Compute the inductive sequence of assertions using the weakest precondition.
4) *LV* Optionally, depending on the choice in the previous step, perform one of the following:
   a) Compute the set of future-live variables for all transitions. Project each predicate in the inductive set of assertions using existential projection.
   b) Compute the set of past-live variables for all transitions. Project each predicate in the inductive set of assertions using universal projection.
5) The output is the predicates of the inductive set of assertions.

Counting the options, there are eight methods in the Newton family that we all implemented in Theta. Architecture-wise, we created a new expression trace checker for all eight kinds. Newton algorithms - similarly to interpolation-based methods - produce a list of predicates for the trace as refutation. Since the Newton algorithms all receive the abstract counterexample as an input and produce a list of predicates as output, no other adjustments were needed.

A short example for the Newton algorithm `NWT_IT_SP` can be seen in Table II. First, the trace formula is computed by unfolding all expressions in the transitions. The unsatisfiability of the trace formula comes from the contradicting values of variable x, so the unsatisfiable core will contain all formulas "writing" x (highlighted with gray). The single transition that did not contribute a formula to the unsatisfiable core is then abstracted to create the abstract trace. Finally, the strongest postcondition operator is used to create an inductive sequence of assertions starting from $true$:

$$\begin{aligned}
\text{post}(x := 255, true) &= (x = 255)\\
\text{post}(x := x + 1, x = 255) &= (x = 256)\\
\text{post}(havoc\ y, x = 256) &= (x = 256)\\
\text{post}([x = 0], x = 256) &= false
\end{aligned}$$

V. Preliminary Evaluation

Our main goal during the evaluation was to test whether Theta can verify problems containing bitvectors, while our secondary objective was to compare the performance of Newton refinement to the preexisting interpolation-based methods.

TABLE II
A short example for `NWT_IT_SP`

Trace	Trace formula	Abstract trace	Inductive sequence of assertions (predicates)
			$true$
$x := 255$	$x_0 = 255$	$x := 255$	
			$x = 255$
$x := x + 1$	$x_1 = x_0 + 1$	$x := x + 1$	
			$x = 256$
$y := 2 * x$	$y_0 = 2 * x_1$	$havoc\ y$	
			$x = 256$
$[x = 0]$	$x_1 = 0$	$[x = 0]$	
			$false$

TABLE III
Number of Successfully Verified Bitvector Problems

Refinement strategy	Small examples	SV-COMP
`NWT_SP`	1	2
`NWT_WP`	3	4
`NWT_IT_SP`	2	4
`NWT_IT_WP`	8	9
`NWT_SP_LV`	1	1
`NWT_WP_LV`	2	1
`NWT_IT_SP_LV`	2	4
`NWT_IT_WP_LV`	5	5
Total examples	15	20

To test the bitvector support, we created multiple smaller examples by hand and converted some of the inputs from the bitvector category of the *Competition on Software Verification (SV-COMP)* [20]. Our results are shown in Table III.

We applied a timeout of 1 hour and considered the problem successfully verified if Theta yielded a correct result inside the timeframe. As it can be seen, all 8 Newton strategies were able to verify some problems, but `NWT_IT_WP` seemed to be the most effective of them.

To find an answer for our secondary goal, we compared the Newton algorithms with an interpolation-based method. We used inputs that did not contain bitvectors, as Z3 cannot interpolate with them. Our inputs included examples from the SV-COMP benchmark set and PLC codes provided by our industrial partners. The results can be seen in Table IV.

The problems were considered successfully verified if Theta provided a correct output in less than 5 minutes. Our baseline

TABLE IV
Number of Successfully Verified Problems

Refinement strategy	Abstraction type: Predicate abstraction	Abstraction type: Explicit value analysis
`SEQ_ITP`	413	203
`NWT_SP`	28	50
`NWT_WP`	260	56
`NWT_IT_SP`	235	230
`NWT_IT_WP`	370	243
`NWT_SP_LV`	30	60
`NWT_WP_LV`	255	54
`NWT_IT_SP_LV`	210	162
`NWT_IT_WP_LV`	242	239
Total examples	548	

was the SEQ_ITP refinement strategy, as it was shown to be an effective strategy earlier [21]. We also differentiated the type of abstraction to use: we considered predicate abstraction [8] and explicit value analysis [9]. All in all, we can see that when using predicate abstraction, SEQ_ITP remained the strongest candidate, and NWT_IT_WP performed best. On the other hand, when comparing the results of explicit value analysis, we can see that three strategies performed better than SEQ_ITP, and among them also NWT_IT_WP was the best.

VI. Conclusion

Our goal was to apply Theta's formal verification methods to safety-critical systems. A major hurdle was that these systems often used language constructs and semantics that Theta did not support.

We introduced bitvector support in Theta, whose semantics conforms to those software that are used in safety-critical (or any computer-based) systems. Introducing bitvector support leads to the need to introduce new refinement algorithms due to the limitation of Theta's underlying SMT solver, Z3. We implemented the Newton-style refinement methods that do not rely on interpolation to be used hand-in-hand with bitvectors.

We also did a small-scale evaluation of the introduced features. We showed that problems with bitvectors could be verified with Theta, and gave evidence that Newton style refinement can indeed outperform the interpolation-based methods, so they are worth to be examined further.

In the future, we will further extend the language support of Theta to provide applicable formal verification for safety-critical software. In addition, we plan to do more extensive measurements to find the strengths and weaknesses of the applied refinement strategies.

References

[1] T. Tóth, A. Hajdu, A. Vörös, Z. Micskei, and I. Majzik, "Theta: a framework for abstraction refinement-based model checking," in *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, D. Stewart and G. Weissenbacher, Eds., 2017, pp. 176–179.

[2] C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Cham: Springer International Publishing, 2018, pp. 305–343.

[3] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[4] C. Barrett, A. Stump, C. Tinelli *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[5] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[6] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to model checking," in *Handbook of Model Checking*. Springer, 2018, pp. 1–26.

[7] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 504–518.

[8] R. Jhala, A. Podelski, and A. Rybalchenko, *Predicate Abstraction for Program Verification*. Cham: Springer International Publishing, 2018, pp. 447–491.

[9] D. Beyer and S. Löwe, "Explicit-state software model checking based on cegar and interpolation," in *Fundamental Approaches to Software Engineering*, V. Cortellessa and D. Varró, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 146–162.

[10] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Program analysis with dynamic precision adjustment," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 29–38.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[12] Á. Hajdu, T. Tóth, A. Vörös, and I. Majzik, "A configurable cegar framework with interpolation-based refinements," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2016, pp. 158–174.

[13] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal—a tool suite for automatic verification of real-time systems," in *International hybrid systems workshop*. Springer, 1995, pp. 232–243.

[14] K. L. McMillan, "Applications of craig interpolants in model checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2005, pp. 1–12.

[15] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[16] H. Barbosa, J. Hoenicke, and A. Hyvarinen, "15th international satisfiability modulo theories competition (smt-comp 2020): Rules and procedures."

[17] D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski, "Craig vs. newton in software model checking," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 487–497. [Online]. Available: https://doi.org/10.1145/3106237.3106307

[18] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2005, pp. 82–87.

[19] R. Grigore, J. Charles, F. Fairmichael, and J. Kiniry, "Strongest post-condition of unstructured programs," in *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, 2009, pp. 1–7.

[20] D. Beyer, "Competition on software verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 504–524.

[21] Á. Hajdu and Z. Micskei, "Efficient strategies for CEGAR-based model checking," *Journal of Automated Reasoning*, vol. 64, no. 6, pp. 1051–1091, 2020.