Using DimSpec for Bounded and Unbounded Software Model Checking

Marko Kleine Büning, Tomáš Balyo, and Carsten Sinz(✉)

Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{marko.kleinebuening,tomas.balyo,carsten.sinz}@kit.edu

Abstract. This paper describes a unified approach for both bounded and unbounded software model checking to find errors in programs written in the programming language C. It is based on a propositional logic intermediate representation, called DimSpec, that has been successfully applied in SAT-based automated planning. Using DimSpec formulas allows us to exploit the advantages of incremental SAT solving and provides an alternative approach to using the universal incremental SAT API IPASIR or native solver APIs. The DimSpec formula can be used for bounded model checking (via incremental SAT solving) as well as unbounded model checking (using a backend that implements an IC3-style algorithm). We also present an implementation of our approach, called LLUMC, which encodes the presence of certain errors in a C program into a DimSpec formula. We evaluate our approach on benchmark problems from the Software Verification Competition (SV-COMP) and compare it with other tools to demonstrate runtime and functionality advantages compared to state-of-the-art solvers.

1 Introduction

A DimSpec formula [30] consists of four CNF formulas $\mathcal{I}$,$\mathcal{U}$,$\mathcal{T}$ and $\mathcal{G}$ which specify a transition system. The formula $\mathcal{I}$ describes the initial state, $\mathcal{G}$ the goal state, $\mathcal{U}$ describes the constraints that must hold in each individual step of the process and finally $\mathcal{T}$ describes the relation of each pair of neighboring steps. DimSpec has been very successfully used for SAT-based automated planning [16]. In this paper we demonstrate that the DimSpec format is also very useful for software verification.

Software has become an important part of almost all modern technical devices, such as cars, airplanes, household appliances, therapy machines, and many more. The cars of tomorrow will drive on their own, controlled by software. As shown by serious accidents like the rocket crash of Ariane flight 501 [24], the massive overdoses of radiation generated by the therapy machine Therac-25 [25] or the car crash of the Toyota Camry in 2005 [22], software is never perfect and almost inevitably contains errors and bugs. While testing of software can,

This work was partially supported by Baden-Württemberg Stiftung within project HIVES.

© Springer Nature Switzerland AG 2019
Y. Ait-Ameur and S. Qin (Eds.): ICFEM 2019, LNCS 11852, pp. 19–35, 2019.
https://doi.org/10.1007/978-3-030-32409-4_2

in practice, only cover a limited number of program executions, software verification can guarantee a much higher coverage while producing proofs for the existence or absence of errors. Many software verification approaches have been developed, for instance symbolic execution [19], (bounded) model checking [9], or abstraction and interpolation [1]. In bounded model checking, function calls are inlined and loops unrolled a finite number of times. This unrolling reduces the complexity of the problem to a computationally feasible level, though it limits coverage and thus precision of the approach.

We developed an approach that is suitable for both bounded and unbounded model checking. To this end, we produce a SAT encoding of a transition system that is general enough to be solved with different solver back-ends, based on, e.g., incremental SAT or on an invariant checking algorithm. We focus on sequential programs written in C, and use the low-level code representation of the compiler framework LLVM as an intermediate language. Based on this representation, we derived an encoding of the program verification task into a DimSpec formula. We first encode the program into four SMT formulas and, subsequently, generate the SAT-level representation in the desired DimSpec format. The resulting formula is then solved by either an incremental SAT solver that unrolls the transition system to find a path to an error state, or an invariant checking algorithm that refines an over-approximation.

Our verification system uses Clang and LLVM version 3.7.1 to compile C-code into LLVM Intermediate Representation. Then our new tool LLUMC (Low-Level Unbounded Model Checker) translates the LLVM-IR representation of the program P to be verified to a DimSpec formula with error states that are reachable iff P contains a corresponding error. To solve the generated formulas we either use the incremental SAT solver IncPlan [16] or the invariant checking algorithm implemented in the solver MinireachIC3 [30]. LLUMC was inspired by the bounded model checker LLBMC [28] but runs independently. Our evaluation is based on the Software Verification Competition (SV-COMP) and shows the correctness and feasibility of our approach. LLUMC is available online at [21].

2 The DimSpec Format

We assume the reader to be familiar with propositional logic, first-order-logic and the Boolean satisfiability problem (SAT), and use definitions and notations standard in SAT. In this section, for completeness, we introduce incremental SAT-solving and describe the theory of bit-vectors in the context of SMT-solving.

Incremental SAT-Solving. Incremental SAT-solving is an approach to solve several related SAT-problems efficiently. In the *assumption based interface* [14], two methods are used to describe a related problem relative to a base problem: `add`(C) and `solve`(A), where C is a clause and A a set of literals called assumptions. Clauses can be added with the `add` method and their conjunction, together with previously added clauses, can then be solved under the condition that all literals in A are true by `solve`(A). To enable simulating the removal of

a clause C between invocations of `solve(A)`, a clause $C' = C \cup a$ is passed to the solver instead of C, with a (called an *activation literal*) being an otherwise unused literal. C is then effectively taken into account iff $\neg a$ is present in A.

DimSpec Formulas. A DimSpec formula [31] represents a transition system with a finite number of states $t_0, t_1, \ldots, t_k$, where each state is a full truth assignment on n Boolean variables $x_1, \ldots, x_n$. It consists of four CNF formulas: $\mathcal{I}, \mathcal{U}, \mathcal{G}$ and $\mathcal{T}$, where $\mathcal{I}$ encodes the set of initial states, $\mathcal{G}$ describes the set of goal states (that in our case indicate occurrence of a program error). Formula $\mathcal{U}$ encodes global constraints that have to hold in each state, and finally the transition clauses $\mathcal{T}$ are satisfied by each pair of consecutive states t_i, t_{i+1}. The clause sets $\mathcal{I}$, $\mathcal{U}$, and $\mathcal{G}$ contain variables $x_1, \ldots, x_n$, and $\mathcal{T}$ contains $x_1, \ldots, x_{2n}$, where $x_1, \ldots, x_n$ encodes the current and $x_{n+1}, \ldots, x_{2n}$ the next state. Testing whether the goal state is reachable from the initial state within k steps is equivalent to checking whether the following formula F_k is satisfiable.

$$F_k = \mathcal{I}(0) \wedge \left(\bigwedge_{i=0}^{k-1} \big(\mathcal{U}(i) \wedge \mathcal{T}(i, i+1) \big) \right) \wedge \mathcal{U}(k) \wedge \mathcal{G}(k),$$

where $\mathcal{I}(i)$, $\mathcal{G}(i)$, $\mathcal{U}(i)$ and $\mathcal{T}(i, i+1)$ denote the respective formulas without index, where each variable x_j is replaced by $x_{j+i \cdot n}$.

DimSpec formulas have been successfully employed in SAT-based automated planning [16,30], but they represent a generic approach to utilize incremental SAT solving for reachability analysis of transition systems. DimSpec solvers can be developed independently of their usage and also be parallelized, which brings benefit to all DimSpec applications.

Incremental SAT Solving for DimSpec. The straightforward way to solve a DimSpec formula is to unroll the transition relation step by step, constructing and solving the resulting formula F_i at each step, until a satisfiable formula is observed. An efficient way to implement this is to use an incremental SAT solver with the assumption-based interface via the following steps:

$$\begin{aligned} \text{step}(0): \quad & \texttt{add}(\mathcal{I}(0) \wedge (a_0 \vee \mathcal{G}(0)) \wedge \mathcal{U}(0)) \\ & \texttt{solve}(\{\neg a_0\}) \\ \text{step}(k): \quad & \texttt{add}(\mathcal{T}(k-1, k) \wedge (a_k \vee \mathcal{G}(k)) \wedge \mathcal{U}(k)) \\ & \texttt{solve}(\{\neg a_k\}) \,. \end{aligned}$$

This algorithm, in practice, only terminates in reasonable time if the goal state is reachable from the initial state. Otherwise it searches "endlessly", i.e. up to a bound of 2^n in the worst case. A more sophisticated approach that can detect unreachability is described next.

IC3 Algorithm. A different approach to solve a DimSpec formula is described in [12] and implemented, among others, in the tool IC3 (**I**ncremental **C**onstruction of **I**nductive **C**lauses for **I**ndubitable **C**orrectness). Given a transition system S and a safety property P, the algorithm can prove that P is S-*invariant*, meaning that, regarding S, property P is true in all reachable states, or produce a counterexample. IC3 incrementally refines a sequence of formulas $F'_0, F'_1, \dots, F'_k$ that describe over-approximations of the set of states reachable in at most k steps. It can extend the formula sequence in major steps that increase k by one. In minor steps the algorithm refines the approximations F'_i with $0 \leq i \leq k$ by conjoining clauses to the F'_i. Given a finite transition system S and a safety property P, the IC3 algorithm terminates and returns true, iff P is true in all reachable states of S [12]. The IC3 algorithm was implemented and adjusted[1] to the DimSpec format in the tool MinireachIC3 by Suda [30].

Comparison to Other SAT Formats. An alternative approach to DimSpec for utilizing the benefits of incremental SAT solving is the IPASIR interface introduced for the 2015 International SAT Race [4]. In contrast to DimSpec, which is a file format, IPASIR is a collection of C/C++ function prototypes, i.e., an application program interface (API). Numerous state-of-the-art SAT solvers implement the IPASIR interface, which makes it very easy and convenient to develop applications using incremental SAT solving without committing to any particular SAT solver.

The advantages of IPASIR over DimSpec are more flexibility (the clauses for the next incremental SAT call can be constructed dynamically based on previous results), more functionality (IPASIR provides much more control over the SAT solver and allows the user to extract more information from the SAT solving process, such as learned clauses or failed assumptions). On the other hand, DimSpec is much easier to use since it does not require any programming and it can be used to express unreachability of transition systems, which is impossible with IPASIR. Furthermore, any SAT solver supporting IPASIR be can used in the IncPlan application [16], which renders it into a DimSpec solver. In summary, DimSpec is a purely declarative approach while IPASIR is procedural.

Another declarative format related to DimSpec is AIGER [10] with safety invariants. AIGER is the format for representing and-inverter graphs, which represent a structural implementation of the logical functionality of a circuit or network. DimSpec and AIGER-safety are mutually translatable[2].

3 Encoding for Software Model Checking

We give a short introduction into the Satisfiability Modulo Theories (SMT) and the LLVM Framework, which are necessary to understand the encoding.

[1] The clause sets $\mathcal{I}, \mathcal{U}, \mathcal{T}$ represent the transition system S, and $\mathcal{G}$ represents the negation of the invariant property P.

[2] We omit the description of these translations due to space limitations.

Afterwards, we will describe a DimSpec encoding for the software model checking approach in more detail to show the feasibility and advantages of encoding problems into the DimSpec format.

Satisfiability Modulo Theories (SMT). Due to quantifiers and infinite domains, first-order-logic is generally undecidable but there are numerous decidable sub-theories. As is for example described in [11], the problem of solving those subsets or theories is called satisfiability modulo theories or SMT. These theories can be seen as restrictions on possible models of first-order-logic formulas [27]. For our encoding, we will only use the theory of bit-vectors. SMT was standardized by the SMT-LIB initiative [5]. We will use the same notations, especially when referring to SMT functions defined in the different theories. Such an SMT-LIB function could for example be $\mathrm{bvadd}(b_1, b_2)$, describing the addition of two bit-vectors b_1 and b_2. A more complex function is called `if-then-else` (ite) and is defined by:

$$\forall c \in BV_1, x, y, z \in BV_i \, (x = \mathrm{ite}(c, y, z) \Leftrightarrow c \wedge x = y \vee \neg c \wedge x = z) \,. \qquad (1)$$

We refer to the *theory of fixed-size bit-vectors* defined by the SMT-LIB standard in [5]. The theory of bit-vectors models finite bit-vectors BV_n of length n and operations on these vectors in first-order-logic. The set of function symbols contains standard operations on bit-vectors such as addition or concatenation.

LLVM Representation. LLVM is an open source compiler framework that consists of a "collection of modular and reusable compiler and tool-chain technologies" [26]. It supports compilation for a wide range of languages and is known for its research friendliness and good documentation. To work directly on C-code is very complex and it is extremely cumbersome to support all language features. Thus, we use the intermediate language of LLVM, which allows for a much simpler characterization of the semantics of statements and provides a number of optimizations and simplifications suitable for our approach. We describe the constructs of LLVM bottom up. The smallest executable unit is called an *instruction*. An instruction is an atomic unit of execution that performs a single operation. A basic block is a linear sequence of program instructions having one entry point and one exit point. It may have multiple predecessors and successors and may also be its own successor. The last instruction of every basic block is called *terminator*. Every basic block is part of a *function*. A function (n, B, e) is a tuple of a name n, a sequence of basic blocks $B = (b_0, b_1, ..., b_m)$, and an entry block $e \in B$. Hereinafter, we will denote the main function of a program with f_{main}. A module $m = (F_m, G_m)$ is a pair of a set of functions F_m (including f_{main}) and a set of global variables G_m.

To optimize our encoding, we run some predefined optimization passes from LLVM and LLBMC on the generated LLVM-module. Among other things, these optimizations handle uninitialized local variables in C-code, promote memory references to register references (as far as possible) and inline all functions into

one main function. These optimizations are described in more detail in [20]. The resulting LLVM-module is then used as input for our encoding.

3.1 Idea and Error Definition

A bug or error in a software program is a well-known notion, but there exists no universal definition. A general concept is that a program has an error, if it does not act according to its specification. For this paper we concentrate on notions standardized in the SV-COMP competitions. Thus, we consider calls to assume and assert and support both standard ANSI-C and notions used in benchmarks of the competition. We state that a program acts according to its specification if the assert statements are true if all assume conditions are met. If an assume condition is not met, the further run of the program is not specified and thus no errors can occur.

Definition 1 (Program Error in LLUMC). *Let P be a program. Then there exists an error in P, if all calls to assume that are prior to an assert statement are true and a call to assert with a parameter value of false is invoked.*

Of course, there are other errors that can happen during a program execution like irregular bit-shifting, non-termination, or integer and buffer overflows.[3]

To verify a C program P with respect to Definition 1, we first translate P to LLVM-IR (i.e. an LLVM-module) using the Clang compiler. After inlining all function calls, we can concentrate on just the main function. Every basic block together with its variable assignment can be seen as a *state*. We then add a special *error state* and try to find a path from the entry state, defined by the entry block of the main function, to the error state.

3.2 State Space

Transitions from one state to a next state will always represent transitions from one basic block to the next with respect to its current variable assignment. Often this kind of encoding is called *small block encoding* [7]. According to the theory of bit-vectors, we define every state variable as a bit-vector of length n. The number of bit-vectors in the state, including the bit-vectors representing the current and previous basic block, define the number of SMT variables that are needed to encode the state. The number of bits in total, i.e. the sum of the length of all bit-vectors encoding a state, equals the number of CNF variables needed.

In our approach, we ignore memory accesses by over-approximating them (i.e. each memory read results in a non-deterministic value). Accesses to stack variables, which in most cases can be put into virtual registers by LLVM, are handled precisely, though, and are sufficient in many cases.[4] First of all, every state has to save the current basic block. Hereinafter, $|B|$ denotes the number of

[3] In our tool LLUMC, we have additionally implemented checks for integer overflows. These are not part of our experimental evaluation, though.

[4] Integrating a full memory model into our approach is part of future work.

basic blocks of the main function after inlining. For our encoding we need two additional blocks. The *ok* block represents a safe state from which no more errors can occur. This block is reached when the program terminates or when an assume condition is not met. The second block is called *error* and is our goal state, representing that an error occurred. With the function $enc(bb) : BasicBlock \rightarrow \mathbb{N}$ we injectively map every basic block to a natural number. If there are $|B|$ basic blocks in *main*, the required length of the bit-vector encoding a state's basic block is $\lceil \log_2(|B| + 2) \rceil$. We call the SMT-variable encoding the current basic block *curr*. In LLVM, the value of a register can depend on the previous basic block (more specifically, this is the case for `phi` instructions) and must thus also be encoded, resulting in another bit-vector of length $\lceil \log_2(|B| + 2) \rceil$, called *pred*. Furthermore, we need to save the current variable assignment. We do not need the assignment of all variables, but should focus on those that will be accessed later on and cannot be eliminated through optimization. Those variables can be classified by two properties. We call the set of those variables V, consisting of

1. variables that are used in more than one basic block and
2. variables that are read before written in a basic block that is part of a loop.

The length of the variables depends on their type. The standard integer type (int) in C has a width of 32 bits on many architectures, long has 64, and Boolean values have a width of 1. There are other types, but their lengths are always specified by LLVM and thus can easily be extracted.

Definition 2 (State). *The state space is the Cartesian product over the set* V^* *of all state variables and the two state-encoding basic-block variables:* $V^* = \{curr,\ pred\} \cup V$. *Every variable* v *of the state space has a fixed bit-length* ℓ_v*. For a specific step* k*, the state* state(k) *is the assignment of concrete bit-vector values to every variable.*

3.3 Encoding to DimSpec Format

Our goal is to encode an LLVM-module as defined at the beginning of this section into DimSpec format. Therefore, we must define the four CNF formulas $\{\mathcal{I}, \mathcal{G}, \mathcal{U}, \mathcal{T}\}$ in such a way that if there exists a transition from $\mathcal{I}$ to $\mathcal{G}$ defined by $\mathcal{T}$ and restricted by $\mathcal{U}$ then there exists an error in the given program code.[5]

The initial formula $\mathcal{I}$ can be created by encoding the entry block of the LLVM-module. The encoding has to represent the state that we are currently at the first basic block and that there were no prior actions. We declare the entry block itself as the predecessor to exclude any prior actions. The entry block and thus the initial formula is independent from any transition. The rest of the variable assignment is arbitrary at this point and can be left undeclared. The encoding of the goal formula $\mathcal{G}$ can be defined accordingly.

[5] A detailed example of the encoding, starting with C-code, over the LLVM representation to the SMT encoding, can be found online at https://baldur.iti.kit.edu/icfem2019/Appendix.pdf.

Definition 3 (Encoding of the Initial and Goal Formula). *Let* entry *be the name of the first block and let* error *be the name of the error block, then the initial formula* $\mathcal{I}(k)$ *and the goal formula* $\mathcal{G}(k)$ *for the LLVM-module and for* $k \in \mathbb{N}$ *are defined as:*

$$\begin{aligned} \mathcal{I}(k) = &\quad curr = \text{enc}(entry) \ \wedge \ pred = \text{enc}(entry), \\ \mathcal{G}(k) = &\quad curr = \text{enc}(error). \end{aligned}$$

The universal formula consists of constraints that have to be true in all states. In our case, that are boundaries for the variables *curr* and *pred*. In the previous section, the number of bits needed to encode the current and previous basic block were defined as $\lceil \log_2(|B| + 2) \rceil$. In most cases $|B| + 2$ is not a power of two and thus bigger numbers can be represented. These numbers must be excluded at all times in the universal formula $\mathcal{U}$.

Definition 4 (Encoding of the Universal Formula). *Let* $|B|$ *be the number of basic blocks in the LLVM-module, then the universal formula* $\mathcal{U}(k)$ *for* $k \in \mathbb{N}$ *is defined as:*

$$\mathcal{U}(k) = \quad curr \leq (|B| + 2) \ \wedge \ pred \leq (|B| + 2)\,.$$

At last, we have to define the transition formula. It represents the transition between state k and state $k+1$. It is important to notice that the transition formula has twice as much variables as the other formulas. To distinguish between the variables in time-point k and $k+1$ every variable v of our state space is called v' at time-point $k + 1$. Otherwise, every transition formula would be evaluated to false and thus no transition step could ever be taken. In general, the encoding of one transition has the form:

$$state(k) \Rightarrow state(k + 1). \tag{2}$$

We call $state(k)$ antecedent and $state(k + 1)$ consequent. For each $state(k)$ that is reachable from our initial state, a transition must be defined. An undefined transition leads to an undefined $state(k + 1)$ with arbitrary values. Thus, if there is a reachable, undefined transition all goal states can be reached. For the same reason, we determine that for each $state(k)$ the transition must be explicit. Variables that are not important for the transition should not be declared in the antecedent but should be specified in the consequent to avoid undefined values. We will use the auxiliary function

$$same(bb) : \text{Basic Block} \rightarrow \text{SMT-formula}$$

to encode that variables which are not modified in a basic block maintain their current value. The function $same(bb)$ returns the conjunction of all $var = var'$, for all variables in our state space, that have not been modified in the transition of our basic block *bb*.

To encode the transition between steps, we take a closer look at the current basic block, further denoted as *bb* and customize Eq. 2 for different branching possibilities. We divide basic blocks into three groups and distinguish them by

means of their terminator. The three different types of terminator instructions are called unconditional branching, conditional branching and return.

Unconditional Branching (br %bb2): Branches to the basic block with the label $bb2$ and creates a transition from the current basic block to $bb2$. If the current basic block has no other instructions, only the change of basic block and the saving of the predecessor have to be encoded. Furthermore, we have to state that no variables have changed during this transition:

$$curr = enc(bb) \Rightarrow curr' = enc(bb2) \wedge pred' = enc(bb) \wedge same(bb). \tag{3}$$

This encoding is rarely complete, because it does not regard all other instructions in the basic block bb. Let rl_{bb} be the ordered list of instructions from bottom to top in bb. Then we iterate over rl_{bb} and regard all instructions $inst$ that are part of our state variables $inst \in V$ and are not the terminator instruction. The instruction is then recursively encoded according to its type and its operands. When an instruction like `%tmp3 = add i32 10 %tmp2` is encoded by the method `visitInst(%tmp3)`, the algorithm checks the operands first. When regarding the value `%tmp2`, the algorithm checks whether it is a variable that is part of our state or a value calculated by an instruction, which the algorithm has to then encode recursively. The stop criterion is always the occurrence of a state variable, a constant or a call to assert, assume or error. The encoding then creates SMT formulas dependent on the operands. Assuming `%tmp2` is a variable from our state space, the encoding for the add instruction would result in `tmp3' = add(10, tmp2)`. This generated SMT formula is then conjuncted with the consequent of Eq. 3. The algorithm continues by iterating further through the list rl_{bb} until there are no instructions left.

Conditional Branching (br %cond, %bb1, %bb2): Creates a transition to $bb1$ with the condition $cond = 1$ and a transition to $bb2$ with the condition $cond = 0$. Every conditional branch has a branching condition represented as a variable ($cond$). We can extract that condition by visiting and encoding the variable representing the branching condition. In LLVM this branching condition is represented as a Boolean value that is assigned by the so called *icmp-instruction*. This instruction returns a Boolean value based on the comparison of two values and it supports equality, unsigned and signed comparison. The icmp-instruction is then encoded recursively by visiting its two operands with the same visiting approach as described for the unconditional branching. The result could for example be the SMT encoding of the mathematical condition $tmp2 > 10$. Based on it, the algorithm creates two separate transitions.

$$\begin{aligned} curr = enc(bb) \wedge visitInst(cond) \Rightarrow \\ curr' = enc(bb1) \wedge pred' = enc(bb) \wedge same(bb). \\ curr = enc(bb) \wedge \neg(visitInst(cond)) \Rightarrow \\ curr' = enc(bb2) \wedge pred' = enc(bb) \wedge same(bb). \end{aligned}$$

Furthermore, the list rl_{bb} is traversed as described previously resulting in a final encoding of the current basic block.

Return Value (*ret val***):** The value *val* can be an arbitrary integer and represents the return value of the program as usual. This terminator creates a transition to *ok*. In an extended and already implemented version, another check is inserted verifying that the result value of a correct program is 0 and if this does not hold a transition to *error* is created.

After encoding branching possibilities, we will look at the calls to assume, assert, error. During the instruction iteration of a basic block, we regard these instructions differently because they lead to a split of our transitions.

Method Calls (Error, Assume, Assert): If the *error*-method, which is used to specify program errors in C-code, is called inside a basic block, we do not have to regard any other instructions and thus delete all other transitions from this basic block. We produce a single transition:

$$curr = enc(bb) \Rightarrow curr' = enc(error) \wedge same(\emptyset).$$

The other three possibilities lead to a split of our transitions similar to the conditional branching. A call of $assume(var)$ divides the set of current transitions for our basic block. The condition is $var = 0$ and leads to a transition to the *ok* state with $s' = enc(ok)$. The call to $assert(var)$ is similar only with the transition to $s' = enc(error)$ if $var = 0$ holds true. In both cases, the encoding continues normally with the next instruction if the conditions are not met.

All components of the transition formula have now been discussed. To obtain the complete transition formula the algorithm has to iterate over all basic blocks of the main function. Depending on their terminator instruction, every basic block has to be encoded according to the definitions above. To predict which transition is taken in which step would be equal to solving the whole formula. Thus, the transition formula is time independent and the transition possibilities for all time steps are part of the formula.

Definition 5 (Encoding of the Transition Formula). *Let BB be the set of all basic blocks of f_{main} and let $encode(b)$ with $b \in BB$ be the encoding as shown above, then the transition formula $\mathcal{T}(k, k+1)$ for $k \in \mathbb{N}$ is defined by:*

$$\mathcal{T}(k, k+1) = \bigwedge_{b \in BB} encode(b). \tag{4}$$

Claim. There exists an error as defined by Definition 1 in program p iff

1. p is transformed into an LLVM-module ℓ as described in Sect. 3 and
2. there exists a transition path in ℓ from the initial state to the goal state while the universal formula holds in all states.

Proof Idea: We forego on a formal proof, because it would require a structural induction over huge sets of C-Code and the LLVM-language. Instead, we present short arguments and references for our claim.

(1): Using LLVM as a representation for C-code is widely accepted and used in research and industry. We assume that the transformation from C-code

into a LLVM-module does not remove or add any errors based on the high number of research papers [1,3,6] and tools like LLBMC [27] and SeaHorn [17].

(2): The error node has two types of incoming edges: from an assert statement and an edge from the error node itself. We disregard the edge that points to itself and are left with the option that match the property defined in Definition 1. If the encoding of the variables is, as we claim, correct and our state space is closed under $\mathcal{T}$ and $\mathcal{U}$, we can assume that the a transition path from the initial state to the error state complies with an error in the LLVM-module.

From SMT to SAT Formula. The encoding of the LLVM-module gives us four SMT formulas. Currently, there are no SMT solver that support the DimSpec format and thus these formulas have to be translated into four CNFs in DimSpec format. The most widespread approach to transform SMT to CNF formulas is called *bit-blasting*. We have taken one approach implemented in STP [15] and the ABC-library [18] and modified these algorithms to correspond to some technical requirements of the DimSpec format. Finally, a CNF in the DimSpec format is created that can be used as input for a number of SAT solvers.

4 Solving the Formula: Bounded vs. Unbounded Model Checking

The general idea of bounded model checking (BMC) is to encode paths of a transition system up to a certain bound. For software, the bound is maintained by unrolling loops and inlining function calls at most k times. The number k is called the *bound* and is the reason for the decidability of bounded model checking but also for its limitations. After the unrolling and encoding of the program, a formula that represents the negation of a desired property is added, and the formula is solved with an SMT or SAT-solver. If the solver finds a model for the formula, the approach has found an error and the model can be used as a counterexample. The loop-bound can be increased step by step until a fixed bound k is reached. The question to which bound the loops should be unrolled is complex and further discussed for example by Biere et al. [9].

As mentioned earlier, our encoding to the DimSpec format leads to a unified encoding for both bounded and unbounded model checking. Whether our approach can be categorized as a bounded or unbounded model checking technique depends on the kind of solver that is used to solve the generated formula.

A first approach is solving the formula with an incremental SAT-solver as described in Sect. 2. We argue that the approach using an incremental SAT-solver has to be categorized as bounded, because the problem is unrolled during solving time and the verification is limited by the number of unrolling steps that can be performed under time and memory restrictions. However, compared with state-of-the-art bounded model checkers, there is a crucial difference in how our verification approach is bounded. Bounded model checkers require a fixed bound

early during their analysis to generate the corresponding problem instance, which cannot be directly reused for other bound settings. For our approach the encoding itself is independent of any unrolling. Only during solving of the instance the loop is unrolled leading to the bound that is perceptible through the time and memory limit which allows us to unroll only a finite number of times.

When solving the generated formula with an invariant checking algorithm as e.g. the in Sect. 2 described IC3-algorithm, the approach becomes unbounded. The whole path to the error label is computed using abstractions which are iteratively refined until either the error path is concrete and no further refinement is possible or a repetition is detected, from which the absence of errors can be deduced. Thus, our approach is truly unbounded, but of course limited by time and memory constraints when solving difficult problems. In summary, our unified encoding can be used for both unbounded and bounded model checking.

5 Experimental Results

The LLUMC-approach is implemented as a tool chain. The input file, a C source file, is compiled with Clang (version 3.7.1) and then optimized with LLVM and LLBMC passes. This optimized LLVM module serves as input for the program LLUMC, which performs the encoding as described above. We modified the tool STP to translate SMT formulas to DimSpec problems. The final renaming and aggregation is implemented directly in LLUMC.

We combined the two different approaches described in Sect. 4 to solve the generated DimSpec/CNF formulas. The tool IncPlan [16] was developed at KIT and implements the incremental SAT-solving interface described in Sect. 3. It can be used with every SAT-solver that accepts the Re-entrant Incremental Satisfiability Application Program Interface (IPASIR). We have evaluated IncPlan with a number of SAT-solvers including Minisat [29], abcdSat [13], Glucose [2] and Picosat [8]. While Glucose and Minisat produced good results for some benchmarks, the IncPlan implementation for these solvers exhibited segmentation-fault errors for some of the benchmark instances. Thus, we focused on the usage of abcdSat and PicoSat. We only show the results of running IncPlan with abcdSat as the backend solver since exchanging abcdSat with PicoSAT resulted in negligible performance differences. For the incremental SAT-solving performed with IncPlan and abcdSat, we are only able to find errors in programs but cannot prove their absence. The reason is the design of the incremental solver IncPlan. It regards the encoding as a path to the error label that has to be found and if there is no such path, the program does not terminate. To also be able to prove the nonexistence of errors an analysis for repetition in the state space has to be performed, which is part of future research.

Secondly, the IC3 algorithm was implemented and adjusted to the DimSpec format in the tool MinireachIC3 [30]. The safety property P expresses that the error state should not be reachable, and thus P is given by $\neg G$, G being the goal formula of the DimSpec encoding. Thus, we are not only able to prove the existence of errors but also their absence.

We ran both tools in parallel and took the results of the tool that terminated first. As both tools are sound, this approach guaranteed the correct result while circumventing disadvantages of each single approach, like the inability to prove the absence of errors through the tool IncPlan. Thus, we are able to take full advantage of the usability of the encoding for different solving techniques.

5.1 Benchmarks

We evaluated our approach using benchmarks from the Software Verification Competition [6]. The SV-COMP is an annual competition for academic software verification tools, with the aim to compare software verifiers. While we did not submit our tool to the competition, the collected benchmarks serve as an excellent evaluation basis for every verifier. All benchmarks are available at [32] and we regarded the sub-folder *c* with programs written in the language C.

The benchmark problem sets are organized by topics. From these benchmarks, we selected all problems compatible with our current LLUMC implementation and thus obtained a total of over 200 problems as a benchmark set. We excluded some benchmarks that included memory accesses or floating point arithmetic. Furthermore, we excluded recursive and concurrent tasks due to the inlining in our approach and thus leaving us with 95 incorrect and 107 correct programs. The benchmarks vary between 14 and 1500 lines of code (LoC).

The evaluation was performed on a system with 64 CPUs with 2.3 GHz and 126 GB working memory. We set a time limit of 600s (wall-clock time) per benchmark problem. We decided to measure the wall-clock-time for the whole LLUMC tool-chain. Due to using GNU parallel [33], we were able to run benchmarks in parallel, but decided to use only 8 CPUs to limit run-time noise arising e.g. due to processes sharing CPU caches. Our approach works sequentially, and parallelism is only achieved by running several benchmarks at once. The DimSpec format supports parallelism on the SAT level, the advantages remain to be evaluated thoroughly in future research.

5.2 Evaluation

We compared our approach to the bounded model checking approach which is implemented for example in the tools CBMC (C Bounded Model Checker) [23] and LLBMC (Low Level Bounded Model Checker) [28]. Both tools, CBMC and LLBMC, are powerful state-of-the-art verification tools, which also earned a number of gold, silver and bronze medals in the SV-COMP competitions.

We created scripts similar to the respective SV-COMP submissions from recent years, but handled some configurations differently. Benchmarking with bounded model checkers requires choosing a suitable loop unroll bound B, resulting in a trade off between precision (increases with B) and speed (decreases with B). For the competition both solvers used specific bounds that were determined through "educated guesses" [23]. Furthermore, in the competition, if a loop-bound was reached and the solver failed to produce an answer, an educated guess was made for the result. In our evaluation, we used the loop unroll bounds

10, 100 and 1000 (in that order), aborting the solving process as soon as a verification result was achieved. When reaching a time or memory limit, we classified the problem instance as *unknown*. The scripts, benchmark sets and detailed results are available at [21].

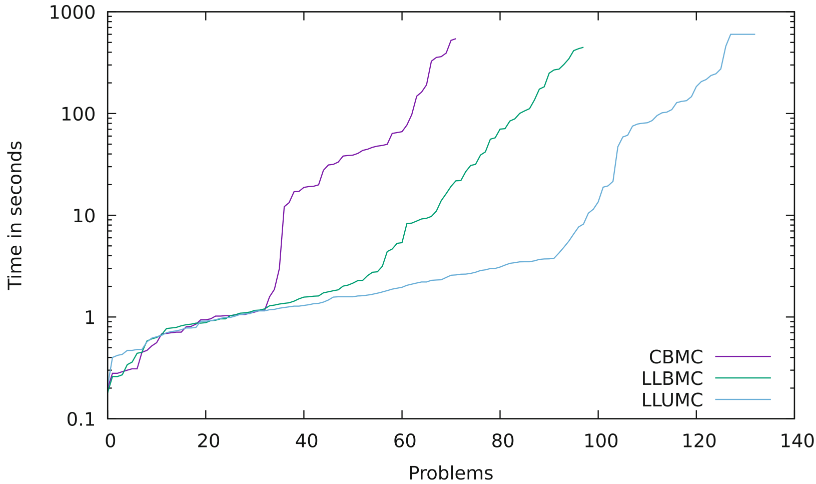

Fig. 1. Comparison of LLUMC with CBMC and LLBMC. The x-axis represents the number of problems the solvers were able to solve and the y-axis the time they needed.

The results of our evaluation are shown in Fig. 1 and indicate both functionality and runtime advantages on the chosen benchmarks[6]. To explain the advantage of our approach and the encoding over the state-of-the-art for bounded model checking, we have to look at the solving approaches individually.

The advantage of the incremental solving with abcdSAT over bounded model checking approaches is caused by our new approach of encoding the verification problem and thereby the bound. With bounded model checking, programs are unrolled to a fixed bound in an early phase of the analysis and the SAT encoding is specifically created for this one bound. This fixed bound is mostly given by the user and when not sufficient enough the verification has to be reattempted for the new bound. With our approach, an unbounded low-level encoding is used, with the unrolling bound being iteratively increased by the incremental backend solver, which is able to reuse facts learned with lower bounds.

The chosen benchmark-set from the SV-COMP includes a large number of problems with unbounded loops and loops with large bounds. While the basic bounded model checking approach cannot handle unbounded loops, the abstraction refinement of MinireachIC3 is able to abstract the state space and prove the absence of errors better then state-of-the-art tools. The number of benchmark

[6] Detailed figures about the single solving approaches can be found online at https://baldur.iti.kit.edu/icfem2019/Appendix.pdf.

problems solved still indicates that proving the absence of errors for programs with large loops is still a difficult task, but the approach using MinireachIC3 leads to a significant improvement.

This experimental evaluation illustrates the feasibility and potential of our approach. We show that our flexible encoding supports a variety of different approaches for solving the generated CNF in DimSpec format. In total, our algorithm is competitive with existing bounded model checkers and can even outperform them on some instances, especially ones with large loop bounds or unbounded loops.

6 Conclusion and Future Work

In this paper we presented the DimSpec format for specifying properties of transition systems on the SAT level. It has already been successfully employed in SAT-based automated planning in the past, and we showed that it can also be advantageous to handle software verification problems. Our new DimSpec-based encoding tool LLUMC can be used to express software verification problems independently from loop-bounds, and thus can be used for both bounded and unbounded model checking. Basing our encoding on DimSpec enables us to leverage powerful DimSpec solvers for software verification.

In future work the performance of the LLUMC approach could be improved by enlarging the incremental steps of the solver. A first evaluation shows that merging basic blocks in LLVM leads to performance improvements, indicating that a large block encoding could be advantageous. Furthermore, adding a full memory model to the LLUMC approach will enable us to support a wider range of C language constructs.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: a framework for abstraction- and interpolation-based software verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_48
2. Audemard, G., Simon, L.: Glucose in the SAT 2014 competition. SAT Compet. **2014**, 31 (2014)
3. Babić, D., Hu, A.J.: Structural abstraction of software verification conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_41
4. Balyo, T., Biere, A., Iser, M., Sinz, C.: SAT race 2015. Artif. Intell. **241**, 45–65 (2016)
5. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-lib standard: version 2.0. In: Proceedings of the 8th International Workshop on SMT, vol. 13, p. 14 (2010)
6. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_43

7. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32. IEEE (2009)
8. Biere, A.: PicoSAT essentials. J. Satisf. Boolean Model. Comput. **4**, 75–97 (2008)
9. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
10. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Technical report, Johannes Kepler University, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
11. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS press, Amsterdam (2009)
12. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
13. Chen, J.: MiniSAT BCD and abcdSAT: solvers based on blocked clause decomposition. SAT RACE (2015)
14. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
15. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_52
16. Gocht, S., Balyo, T.: Accelerating SAT based planning with incremental SAT solving. In: International Conference on Automated Planning and Scheduling (2017)
17. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: a framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 447–450. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_41
18. Jha, S., Limaye, R., Seshia, S.A.: Beaver: engineering an efficient SMT solver for bit-vector arithmetic. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 668–674. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_53
19. Khurshid, S., PĂsĂreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_40
20. Kleine Büning, M.: Unbounded Software Model Checking with Incremental SAT-Solving. Master Thesis at the Karlsruhe Institute for Technology (2017)
21. Kleine Büning, M.: LLUMC (Low Level Unbounded Model Checker (2019). https://github.com/MarkoKleineBuening/LLUMC-Publications
22. Koopman, P.: A case study of Toyota unintended acceleration and software safety. Carnegie Mellon University Presentation, September 2014
23. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
24. Le Lann, G.: An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In: Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems, 1997, pp. 339–346 (1997)
25. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. Computer **26**(7), 18–41 (1993)

26. The LLVM Compiler Infrastructure. http://llvm.org/. Accessed Nov 2018
27. Merz, F.: Theory and Implementation of Software Bounded Model Checking. Ph.D. thesis, Dissertation, Karlsruher Institut für Technologie (KIT) (2016)
28. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. Verified Software: Theories, Tools, Experiments (2012)
29. Sorensson, N., Een, N.: An Extensible SAT-solver. In: 6th International Conference of the Theory and Applications of Satisfiability Testing, SAT 2003, Santa Margherita Ligure, Italy, 5–8 May 2003, pp. 502–518 (2003)
30. Suda, M.: Property directed reachability for automated planning. J. Artif. Intell. Res. (JAIR) **50**, 265–319 (2014)
31. Suda, M.: Dimspec, a format for specifying symbolic transition systems (2016). http://forsyte.at/dimspec
32. SV-Benchmarks. https://github.com/sosy-lab/sv-benchmarks/. Accessed 01 Nov 2018
33. Tange, O., et al.: Gnu parallel-the command-line power tool. USENIX Mag. **36**(1), 42–47 (2011)