Symbolic Computation via Program Transformation*

Henrich Lauko, Petr Ročkai, and Jiří Barnat

{xlauko1,xrockai,barnat}@fi.muni.cz

Abstract. Symbolic computation is an important approach in automated program analysis. Most state-of-the-art tools perform symbolic computation as interpreters and directly maintain symbolic data. In this paper, we show that it is feasible, and in fact practical, to use a compiler-based strategy instead. Using compiler tooling, we propose and implement a transformation which takes a standard program and outputs a program that performs semantically equivalent, but partially symbolic, computation. The transformed program maintains symbolic values internally and operates directly on them hence the program can be processed by a tool without support for symbolic manipulation.

The main motivation for the transformation is in symbolic verification, but there are many other possible use-cases, including test generation and concolic testing. Moreover using the transformation simplifies tools, since the symbolic computation is handled by the program directly. We have implemented the transformation at the level of LLVM bitcode. The paper includes an experimental evaluation, based on an explicit-state software model checker as a verification backend.

1 Introduction

It is common to use symbolic methods in program analysis and verification and related disciplines. Symbolic execution has found numerous use cases in test generation and concolic testing and is widely deployed in practice. Likewise, many modern software verification tools are based on bounded model checking, which combines symbolic execution with SMT solvers to successfully attack hard problems in their problem domain.

On one hand, multiple production-quality SMT solvers are readily available and even provide a common interface [4]. While a certain degree of integration is required to achieve optimal performance, solvers have attained nearly commodity status. This is in stark contrast to symbolic interpretation, which is usually implemented ad-hoc and is not re-usable across tools at all. The only exception may be KLEE [11], a symbolic interpreter for LLVM bitcode, which is used as a backend by a few analysis tools. Undoubtedly, the fact that it is based on the (ubiquitous) LLVM intermediate language has helped it foster wider adoption.

* This work has been partially supported by the Czech Science Foundation grant 18-02177S and by Red Hat, Inc.

Arguably, interpreters (virtual machines) for controlled program execution, as required by dynamical analysis tools, are already complex enough, without involving symbolic computation. To faithfully interpret real-world programs, many features are required, including an efficient memory representation, support for threads, exceptions and a mechanism to deal with system calls. Complexity is, however, undesirable in any system and even more so in verification tools.

For these reasons, we propose to lift symbolic computation into a separate, self-contained module with minimal interfaces to the rest of the verification or analysis system (see Figure 1). The best way to achieve this is to make it *compilation-based*, that is, provide a transformation that turns ordinary (explicit) programs into symbolic programs automatically. The transformed program only uses explicit operations, but it uses them to manipulate symbolic expressions and as a result can be executed using off-the-shelf components.

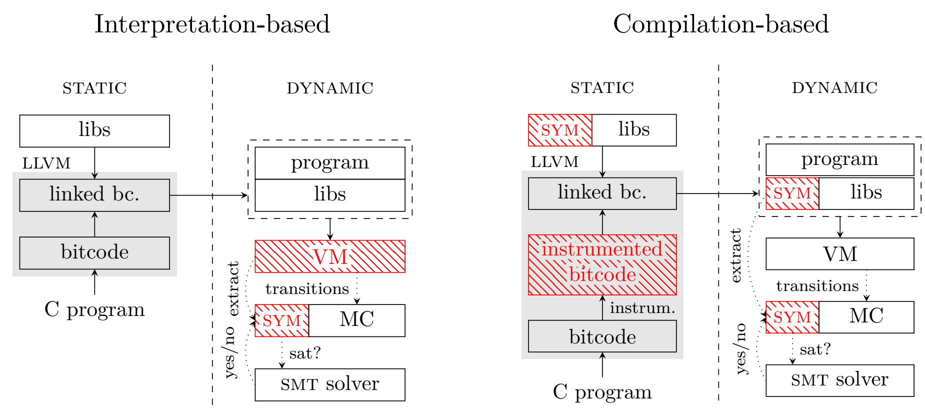

Fig. 1. Comparison of interpretation-based and compilation-based symbolic methods in the context of LLVM model checking. VM stands for 'virtual machine', while MC stands for 'model checker'. The hatched boxes represent components that work with symbolic data. In the compilation-based method, symbolic operations are instrumented into the bitcode, and their implementation is provided in the form of a library. The virtual machine does not need to know about symbolic values at all. The model checker, however, extracts symbolic data and a path condition from the executed program.

The expected result is that the proposed transformation can be combined with an existing solver and a standard explicit interpreter of LLVM bitcode. Depending on how one combines those ingredients, they will obtain different analysis tools. As an example, in Section 5.3, we use the transformation, an existing explicit-state model checker DIVINE and an SMT solver STP [16] to build a simple control-explicit, data-symbolic (CEDS) [5] model checker. Building a tool which implements symbolic execution would be even simpler.[1]

[1] In fact, a control-explicit, data-symbolic model checker already contains a subroutine (in our case about 200 lines) which effectively implements a symbolic executor.

1.1 Goals

Our primary goal is to design a self-contained program transformation that can be used in conjunction with other components to piece together symbolic analysis and verification tools. We would like the transformation to exhibit the following properties:

1. allow mixing of explicit and symbolic computation in a single program,
2. expose a small interface to the rest of the system, and finally
3. impose minimal run-time overhead.

The first property is important because it often does not make sense to perform all computation within a program symbolically. For instance, a symbolic execution engine may wish to natively perform library calls requested by the program. Therefore, it ought to be possible to request, from the outset, that a particular value in the program is symbolic or explicit.

It is unfortunately not possible to execute the symbolised program in a context that is completely unaware of symbolic computation. However, the requirements imposed on the execution environment can be minimised and defined clearly (see Section 5.4). Finally, exploring all possible executions given a single input sequence is already expensive and when used in the context of model checking, we would like to incur as small a penalty as possible.

1.2 Contribution

The idea that various tasks can be shifted between compile time and run time is as old as higher-level programming languages. In the context of verification, there is a large variety of approaches that put different tasks at different points between these two extremes. Symbolic computation is traditionally found near the *interpretation* end of the spectrum.

Our contribution is to challenge this conventional wisdom and show that this technique can be shifted much farther towards the *compilation* end. Further, by treating symbolic computation as an *abstract domain*, we pave the way for other abstract domains to be approached in this manner. Finally, all relevant source code and benchmark data is freely available online.[2]

2 Related Work

Program verification techniques based on symbolic execution [19], symbolic program code analysis [1] and symbolic approach to model checking [22] have been the subject of extensive research.

As for symbolic execution, the approach most closely related to ours is represented by the KLEE symbolic execution engine [11] that performs symbolic

[2] `https://divine.fi.muni.cz/2018/sym`

execution on top of LLVM IR [21]. Besides standalone usage as a symbolic executor, KLEE has become also a back-end tool for other types of analyses and for verification. For example, the tool Symbiotic [13] combines code instrumentation and slicing with KLEE to detect errors in C programs.

Besides symbolic execution, other forms of abstract interpretation, like predicate abstraction, is often used in code analysis. The most successful approaches are based either on counterexample-guided abstraction refinement (CEGAR) [14] or lazy abstraction with interpolants [3], which are implemented in tools such as BLAST [9] and CPAchecker [7]. There are numerous research results in this direction, summarised in e.g. [8, 27, 28].

A verification algorithm that goes beyond static program code analysis and combines predicate abstraction with concrete execution and dynamic analysis has been also introduced [15]. This approach can successfully verify programs that feature unbounded loops and recursion, unlike standard symbolic execution.

Using instrumentation (as opposed to interpretation) for symbolic verification was proposed a few times, but the only extant implementation that works with realistic programs is derived from the CUTE [26] family of concolic testing systems, i.e. the tools CREST [10] and jCUTE [25]. In particular, CREST uses the CIL toolkit[3] to insert additional calls into the program to perform the symbolic part of concolic execution. The approach as described in [26] is limited to symbolic computation, unlike the present paper, which works with arbitrary abstract domains.

A related, process was proposed by Khurshid et al. [18]: in this case, hand-annotated code was processed by Java PathFinder [17], an explicit state model checker. Our approach, in contrast, is fully automatic and more general.

Finally, besides symbolic code analysis and symbolic execution, there are approaches that perform symbolic model checking as such. The key differentiating aspect of symbolic model checking is the ability to decide equality of symbolically represented states. This is important in particular for verification of parallel and reactive programs where the state space contains diamonds or loops, respectively. The tool SymDIVINE [23] is focused on bit-precise symbolic model checking of parallel C and C++ programs. It extends standard explicit state space exploration with SMT machinery to handle non-deterministic data values. As such, SymDIVINE is halfway between a symbolic executor and an explicit-state model checker. Unlike the solution presented in this paper, SymDIVINE does not separate the symbolic interpreter from the core of the model checker. In general, symbolic model checking is more often used with synchronous systems, for example [12].

[3] CIL is short for C Intermediate Language [24], and is a simplified subset of the C language. The toolkit can automatically translate standard C into the intermediate (CIL) form. The CIL form can be optionally brought into the form of three-address code and this feature is used in CREST.

3 Abstraction as a Transformation

While in the present work, our main goal is to transform a concrete program into one that performs symbolic computation, it is expedient to formulate the problem more generally. We will think in terms of an *abstraction*, in the sense of abstract interpretation, which has two main components: it affects how *program states* are represented and it affects the *computation of transitions* between those states. There are two levels on which the abstraction operates:

1. the static level, concerning syntactic constructs and the type system
2. dynamic, or semantic, which concerns actual execution and values

In the rest of this section, we will define *syntactic abstraction* (which covers the static aspects) as means of encoding abstract semantics into a concrete program. While it is convenient to think of the transformed program in terms of abstract values and abstract operations, it is also important to keep in mind that at a lower level, each abstract value is concretely represented (encoded). Likewise, abstract operations (instructions) are realised as sequences of concrete instructions which operate on the concrete representation of abstract values (see Figure 4, left). Those considerations are at the core of the second, dynamic, aspect of abstraction. Reflecting this structure, the program transformation therefore proceeds in two steps:

1. the input program is (syntactically) abstracted
   - concrete values are replaced with abstract values
   - concrete instructions are replaced with abstract instructions
2. abstract instructions are replaced by their concrete realisation

The remainder of this section is organised as follows: in Section 3.1, we describe the expected concrete semantics of the input program. Section 3.2 then introduces syntactic abstraction, Section 3.3 deals with representation and typing of values in the abstracted program, Section 3.6 goes on to describe the treatment of instructions. Section 3.7 briefly discusses interactions of multiple domains within a program and finally Section 3.8 gives an overview of relational abstract domains.

3.1 States and Transitions

We are interested in general programs, e.g. those written in the C programming language. Abstraction is often described in terms of *states* and *transitions*. In case of C programs, a state is described by the content of memory (including registers). Transitions describe how a state changes during computation performed by a given program. In this paper, we will use small-step semantics, partly because the prototype implementation is based on LLVM,[4] and in part because it is a natural choice for describing parallel programs.

[4] Programs in LLVM are in a partial SSA form, a special case of three-address code [2]. Three-address code is essentially small-step semantics in an executable form.

In this description, the transitions between program states are given by the effect of individual instructions on program state. Which instruction is executed and which part of the program state it affects is governed by the source state. Our discussion of *abstract transitions* will therefore focus on the effects of instructions: as an example, the `add` instruction obtains two values of a specified bit width from some locations in the program state, computes their sum and stores the result to a third location.

3.2 Syntactic Abstraction

The input program is given as a collection of functions, each consisting of a control flow graph where nodes are basic blocks – each a sequence of non-branching instructions. Memory access is always explicit: there are instructions for reading and writing memory, but memory is never directly copied, or directly used in computation. While this further restricts the semantics of the input program, it is not at the expense of generality: programs can be easily put in this form, often using commodity tools.

With these considerations in mind, the goal of what we will call *syntactic abstraction* is to replace some of the concrete instructions with their abstract counterparts. The general idea is illustrated in Figure 2.

`x : int ← input()`
`y : int ← factorial(7)`
`z : int ← add(x, y)`
`b : bool ← leq(y, z)`
`assert(b)`
	`x : a_int ← lift(*)`
	`y : int ← factorial(7)`
	`z : a_int ← a_add(x, y)`
	`b : a_bool ← a_leq(y, z)`
	`assert(b)`

Fig. 2. An example of syntactic abstraction. In this example, `a_int` and `a_bool` represent abstract types (see also Section 3.3). We create the abstract value `x` with a `lift(*)` operation to represent an arbitrary value of type `int` (see Section 3.4). Also, notice that the concrete computation of `factorial(7)` remains intact.

Apart from a few special cases, an abstract instruction takes abstract values as its inputs and produces an abstract value as its result. The specific meaning of those abstract instructions and abstract values then defines the *semantic abstraction*. The result of *syntactic* abstraction being performed on the program is, therefore, that the modified program now performs abstract computation. In other words, the transformed program directly operates on abstract states and the effect of the program on abstract states defines the abstract transition system.

We posit that syntactic abstraction, as explained in following sections, will automatically lead to a good semantic abstraction – i.e. one that fits the standard definition: a set of concrete states can be mapped to an abstract state, an abstract state can be realised as a set of concrete states and those operations are compatible in the usual sense.

3.3 Abstract Values and Static Types

A distinguishing feature of the syntactic approach to abstraction is that it admits a static type system. In other words, the variables in the program can be assigned consistent types which respect the boundary between abstract and concrete values. While a type system is a useful consistency check, its main importance lies in facilitating a description of how syntactic abstraction operates.[5]

We start by assuming existence of a set of concrete scalar types, S, and of concrete pointer types. We define a map Γ that builds up all relevant types from the set of scalar types. The set of all types $\Gamma(T)$ derived from a set of scalars T is defined inductively as follows:

1. $T \subseteq \Gamma(T)$, that is, each scalar type is included in $\Gamma(T)$
2. if $t_1, ..., t_n \in \Gamma(T)$ then also the product type $(t_1, ..., t_n) \in \Gamma(T)$, $n \in \mathbb{N}$
3. if $t_1, ..., t_n \in \Gamma(T)$ then also the disjoint union $t_1|t_2|...|t_n \in \Gamma(T)$, $n \in \mathbb{N}$
4. if $t \in \Gamma(T)$ then $t* \in \Gamma(T)$ ($t*$ denotes a pointer type)

In other words, the set $\Gamma(S)$ describes finite (non-recursive) algebraic types over the set of concrete scalars and pointers.

A fundamental building block of the syntactic abstraction is a bijective map α_i, defined for each abstract domain separately,[6] from the set of concrete scalar types S to the set of abstract scalar types $A_i = \alpha_i(S)$ (we let A be the union of all the A_i: $A = A_1 \cup A_2 \cup ...$). Each value which exists in the abstracted program then belongs to a type in $\Gamma(S \cup A)$ – in other words, values are built up from concrete and abstract scalars.

In particular, this means that the abstraction works with *mixed types* – products and unions with both concrete and abstract fields. Likewise, it is possible to form pointers to both abstract values and to mixed aggregates.

3.4 Semantic Abstraction

The maps α_i and α_i^{-1} let us move from concrete to abstract scalar *types* (and back) and are strictly a syntactic construct. The *semantic* (dynamic) counterpart of α_i are lift_i and lower_i: these are not maps, but rather abstract operations (instructions). Just as α_i and α_i^{-1} translate between concrete and abstract types, lift_i goes from concrete to abstract *values* and lower_i the other way around. While both the α_i and lift_i and lower_i are defined on scalar types S and scalar values respectively, they can be all naturally extended to the set of all types $\Gamma(S)$ (and their corresponding values).

[5] Additionally, since the SSA portion of the LLVM IR is already statically typed, we can take advantage of this existing type system in the implementation. Nonetheless, the treatment in this section does not depend on LLVM and would be applicable to any dataflow-oriented program representation.

[6] Since multiple abstract domains can co-exist in a single program, we use the lower index i to distinguish them.

3.5 Representation

Besides α_i, there is another type map, which we will call ρ_i which maps each abstract scalar type in A_i to a concrete type in $\Gamma(S)$. This is the *representation map*, and describes how abstract values are *represented* at runtime. This is to emphasise that abstract values are, in the end, encoded using concrete values that belong to particular concrete types. Moreover, in general for $t \in \Gamma(S)$, $\rho_i(\alpha_i(t)) \neq t$: the representation is unrelated to the original concrete type. An abstract floating point number may be, for instance, represented by a concrete pointer to a concrete aggregate made of two 32-bit integers.

`a : pointer ← malloc(4)`	`a : pointer ← malloc(4)`
`y : a_int ← lift(*)`	`y : term ← sym_lift(*)`
`store y → a`	`freeze y → a`
`...`	`...`
`z : a_int ← load a`	`z : term ← thaw a`

Fig. 3. Freezing and thawing of values transfers them between abstract representation and their concrete realisation. In this case, ρ sends `a_int` to `term`, which realises the term domain described in section Section 4. The *freeze* and *thaw* operations allow `term` to be bigger than the original 4-byte integer type.

While $lift_i$ and $lower_i$ are the value-level counterparts of the map α_i, we need another pair of operations to accompany the representation map ρ_i. We will call them $freeze_i$ and $thaw_i$, and they map between $t \in A_i$ and $\rho_i(t) \in \Gamma(S)$. The idea is that memory manipulation (and manipulation of any concrete aggregates) is done entirely in terms of the representation types (using frozen values), but abstraction operations on scalar values are defined in terms of the abstract type (i.e. thawed values). The use of freezing and thawing is illustrated in Figure 3.

One challenge in the implementation of $freeze_i$ and $thaw_i$ is that the memory layout of a program should not change[7] as a side-effect of the transformation. This means that for many abstract domains, the *freeze* operation must be able to store additional data associated with a given address, and *thaw* must be able to obtain this data efficiently. While this is an implementation issue, it is an important part of the interface between the transformed program and the underlying execution or verification platform. However, since the program is

[7] The exact layout of data (structures, arrays, dynamic memory) is normally the responsibility of the program itself, more so in the case of intermediate or low-level languages. For this reason, it is often the case that the program will make various assumptions about relationships among addresses within the same memory object. It is impractical, if not impossible, to automatically adapt the program to a different data layout, e.g. in case the size of a scalar value would change due to abstraction.

transformed as a whole, there is no need to explicitly track this additional data at runtime.[8]

An additional role of the *freeze*/*thaw* pair is to maintain dynamic type information at runtime. While it is easy to assign static types to instruction operands and results, this is not true for memory locations: different parts of the program can load values of different static types from the same memory address. For this reason, the type system which governs memory use must be dynamic and allow dispatch on the actual (runtime) type of the value stored at a given memory location.

3.6 Abstract Instructions

As indicated at the start of this section, it is advantageous to formulate the transformation in two phases, using intermediate abstract instructions. Abstract instructions take abstract values as operands and give back abstract values as their results. It is, however, of crucial importance that each abstract instruction can be realised as a suitable sequence of concrete instructions. This is what makes it possible to eventually obtain an abstract program that does not actually contain any abstract instructions and execute it using standard (concrete, explicit) methods.

In the first (abstraction) phase, concrete instructions are replaced with their abstract versions: instruction `inst` with a type $(t_1, ..., t_n) \to t_r$ is replaced with `a_inst` of type $(\alpha(t_1), ..., \alpha(t_n)) \to \alpha(t_r)$. Additionally, *lift*, *lower*, *freeze* and *thaw* are inserted as required.[9] The implementation is free to decide which instructions to abstract and where to insert value lifting and lowering, as long as it obeys typing constraints. The specific approach we have taken is discussed in Section 3.7 and the implementation aspects are described in Section 5.2.

After the first phase is finished, the program may be further manipulated in its abstract form before continuing the second phase of the abstraction. This gives a practical, implementation-driven reason for performing the abstraction transformation in two steps, in addition to the conceptual clarity it provides.

In the second step, all abstract operations, including *lift* and *lower*, are realised using concrete subroutines. The realisation (implementation) of `a_inst` is of the type $(\rho(\alpha(t_1)), ..., \rho(\alpha(t_n))) \to \rho(\alpha(t_r)))$, clearly obviating the need for thawing and re-freezing the value.

3.7 Abstract Domains

Necessarily, in an abstracted program, the values it manipulates will come from at least two different domains: the concrete domain and the chosen abstract do-

[8] The only way a value can be copied from one memory address to another is via a `load` instruction followed by a `store`, both of which are instrumented and as such also transfer the supplementary data.

[9] For instance, concrete operands to abstract operations are lifted, arguments to necessarily concrete functions (e.g. real system calls) are lowered. Memory stores are replaced with *freeze* and loads with *thaw*.

main, in line with the first requirement laid out in Section 1.1. This is because it is usually impractical to abstract *all* values that appear in the program. Additional abstract domains, therefore, do not pose any new conceptual problems.

For the sake of simplicity, we only consider instructions where all operands come from the same domain (this holds for both the concrete and for abstract domains). Moreover, the only instructions where the domain of the result does not agree with the domain of the operands are cross-domain conversion operations, which take care of transitioning values from one domain to another. The two most important instances of those operations are *lift* and *lower*[10] introduced in Section 3.3.

```
enum parity
  even, odd, undef

parity_add(a: parity, b: parity)
  if a is undef ∨ b is undef
    return undef
  if a is even: return b
  if b is even: return a
  return even # odd + odd
```

```
lifter_add(x: ?, y: ?)
  if x is int ∧ y is int
    return add(x, y)
  if x is int
    x : a_int ← lift(x)
  else # y is int
    y : a_int ← lift(y)
  return a_add(x, y)
```

Fig. 4. Left: Domain implementation can be provided in a high-level language (e.g. C++) and needs to provide a representation of abstract scalar values and operations on them. An abstract value (of type `parity`) can be `even`, `odd` or in a superposition of those (`undef` – unknown). The term domain described in Section 4 is constructed analogously. Right: A lifter ensures that both arguments to an operation are in the same domain.

Even though cross-domain conversions are necessary in the program, it is a major task of the proposed transformation to minimise their number. A natural approach that would minimise unwanted domain transitions is to propagate abstract domains along the data flow of the program. That is, if an abstract instruction i is already in the program and its result a is also used as an operand elsewhere, we prefer to lift all the users of a into the abstract domain of i (cf. Figure 4, right), instead of lowering a into a set of concrete values. This simple technique, which we call *value propagation*, forms the core of our entire approach (see also Figure 2). It is worth noting that this is particularly simple to do for programs in (partial) SSA form[11], although the variables which are not part of

[10] The names *lift* and *lower* allude to the relationship of the abstract and the concrete domain. In applications with multiple abstract domains, it may be expedient to include additional instructions that convert directly from one abstract domain to another, although in theory it is always possible to go through the concrete domain.

[11] Again, this is true of LLVM bitcode – it is already in a partial SSA form. This simplifies our prototype implementation somewhat.

SSA are still somewhat challenging. Those are covered by the *freeze* and *thaw* operations, which are discussed in more detail in Section 5.1.

Given the above, a logical starting point is to pick an initial set of instructions that we wish to lift into an abstract domain. Those could be explicit *lift* instructions placed in the program by hand, they could be picked by static analysis, or could be the result of abstraction refinement. The abstract program can be then obtained by applying value propagation to this initial set of abstract instructions.

3.8 Constraints and Relational Domains

The last important aspect of abstraction is its effect on control flow of the program. It is often the case that control flow depends on specific values of variables via conditional branching. The condition on the branch is typically a predicate on some value, or a relationship among multiple values that appear in the program. If the involved values are, in fact, abstract values, it is quite possible that *both* results of the predicate or comparison are admissible and that the conditional branch can therefore go both ways. The way we deal with this in the transformation is that the program makes a *nondeterministic choice* on the direction of the branch. How this nondeterministic choice is implemented is again deferred to the execution environment. In any case, the choice of direction provides additional information – constraints – on the possible values of variables (cf. Figure 6).

We encode those constraints into *assume* instructions: given an abstract value and the constraint, *assume* computes the constrained value. Additionally, depending on the abstract domain, it may be desirable to constrain values other than those directly involved in the comparison. Alternatively, relational domains may be able to encode constraint information themselves: this is in particular the case in the *term domain* which realises symbolic computation. Therefore, for the purposes of the present paper, simply inserting a single *assume* instruction on each outgoing edge of the conditional is sufficient.

3.9 Summary

In the above, we have set up abstraction in such a way that it fits into a transformation-based approach. In particular, we have separated *syntactic* and *semantic* abstraction and shown how the former induces the latter. The proposed syntactic abstraction captures how the program is changed, while semantic abstraction captures the dynamic (execution) aspects of abstract interpretation.

4 Symbolic Computation

Now that we have described how to perform program abstraction as a transformation, the remaining task is to re-cast symbolic computation as an abstract domain. Fortunately, this is not very hard: the abstract values in the domain are *terms*, while the abstract instructions simply construct corresponding terms from their operands. In other words, symbolic computation is realised by a *free algebra* (that is, the *term algebra*). The *input values* of the program correspond to nullary symbols – in practice, a unique nullary symbol is created each time the program obtains a value from its input. All the remaining values are built up as terms of bit-vector operations and constants. We will refer to the abstract domain thus formed as the *term domain*.

It is not hard to see that a program transformed this way will simply perform part of its computation symbolically in the usual sense. Additionally, as the computation progresses, *assume* instructions impose a collection of *constraints* on the nullary symbols of the abstract algebra (i.e. the input values). Each constraint takes the form of a term with a relational symbol in the root position. These constraints become part of the abstract state, effectively ensuring that the term domain is fully relational.[12]

It is a requirement of abstract interpretation that it is possible to construct an abstract state from a set of concrete states. In the *term domain* this can be achieved by assigning, to each memory location that differs in some of the concrete states[13], a fresh nullary symbol. We then impose constraints that ensure that exactly the input set of concrete states is represented by the resulting abstract state. For instance, if the input set of concrete states differs by the value of a single variable a, and this variable takes values 1, 2, 3 and 4 in the 4 input states, a suitable constraint would be $a \geq 1 \wedge a \leq 4$.

In some cases, it is impossible to construct the requisite constraints using only conjunction and relational operators. To ensure that the term domain forms a lattice (in particular that a least upper bound always exists), it is necessary to allow the constraints to use logical disjunction.

While the above considerations regarding constraints are an important part of the theoretical underpinnings of the approach, it is almost always entirely impractical to shift back and forth between concrete and abstract states. In practice, therefore, the constraints described in this section simply arise through the *assume* mechanism described in Section 3.8. As such, the constraints that appear in a given state form a *path condition*. Finally, we note that the least upper bound of abstract states defined above corresponds to path conditions which arise from *path merging* in symbolic execution.

5 Implementation

We have implemented the proposed program transformation on top of LLVM, using its C++ API. Both the transformation and all additional code (model checker and solver integration) was done in C++. The transformation itself is the largest component, totalling 3200 lines of code.

[12] An abstract domain is called *relational* when it is capable of preserving information about relationships among various abstract values that appear in the program.

[13] In the present paper, we only deal with abstract (symbolic) *values*. The structure of the program state, that is, the arrangement of the program memory, is taken to be always represented explicitly, i.e., it belongs squarely to the concrete domain.

5.1 Freeze and Thaw

As mentioned in Section 3.7, our implementation is based on the simple idea of maximum propagation of abstract values along the data flow of the program. While the SSA part of the algorithm is essentially trivial, storing abstract values in program memory is slightly more challenging. The purpose of *freeze* and *thaw* is to overcome this issue.

While the dynamic type system that *freeze* and *thaw* provide to the transformed program and the ability to store additional data associated with a given memory address are largely orthogonal at the conceptual level, they are closely related at the level of implementation. This is because in principle, a dynamic type system only requires that additional information is attached to values manipulated by the program, and that this information is correctly propagated. Since apart from memory access, the program is statically typed, it is sufficient to perform dynamic type checks (and dispatch) when a value is *thawed*, while *freeze* simply stores the incoming static type.

Implementation-wise, our target platform is a virtual machine with provisions for associating user-defined metadata to arbitrary memory addresses. This makes the implementation of *freeze* and *thaw* simple and efficient. However, in case such a mechanism is not available, it is sufficient to implement an associative map, using addresses as keys, inside the program.

5.2 Domains

In real-world programs, there are often variables which do not benefit from abstraction or from symbolic treatment, and are best represented explicitly. For this reason, the toplevel abstract domain that we use is the disjoint union (i.e. the type-level sum) of the concrete domain and the term domain. If we denote the concrete domain with C and the symbolic (term) domain with S, the type toplevel type is $C \sqcup S$.

Since the *freeze* and *thaw* operations maintain dynamic type information in the executing program, it is possible to quickly compute operations for which both operands are concrete (explicit). If both operands are symbolic, a symbolic operation is directly invoked, while in the remaining case – one symbolic and one concrete argument – the concrete argument is lifted into the symbolic (term) domain. The procedure is called a *lifter* and is automatically synthesized for each abstract operation that appears in the program. An example of a lifter is given in Figure 4 (right).

It is also possible to use the domain $C \sqcup (C \times S)$, which corresponds to concolic execution (i.e. it maintains both a concrete and a symbolic value at the same time). This requires the additional provision that *assume* instructions obtain concrete values that satisfy the symbolic constraints on their abstract counterparts (an SMT solver will typically provide a model in case the assumptions were feasible, which can then be used to reconstruct the requisite concrete values).

```
a : pointer ← malloc(4)
w : a_int ← lift(*)
x : a_int ← lift(*)
y : a_int ← a_add(w, x)
z : a_int ← a_mul(y, 7)
freeze z → a
```

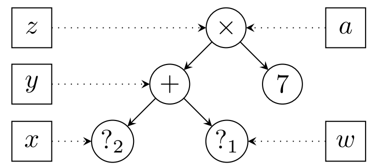

Fig. 5. Example of a formula tree as generated by the term domain. The boxes correspond to abstract variables, while the circles are the concrete representation of terms. Question marks denote unconstrained nullary symbols.

```
x : a_int ← lift(*)
if nondet()
  x': a_int ← assume(x < 10)
  y : a_int ← a_add(x', 1)
else
  x': a_int ← assume(x >= 10)
  y : a_int ← a_sub(x', 1)
```

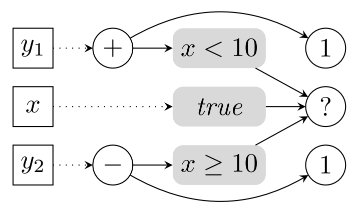

Fig. 6. The program on the left arises from instrumentation of conditional branching, in this case `if x < 10`. The formula tree on the right includes constraints arising from the *assume* instructions. Note that on any given path through the program, only one of the subtrees rooted in y_1 or y_2 can exist.

5.3 Execution & Model Checking

We represent the terms described in Section 4 by a simple tree data structure. The abstract instructions that correspond to operations on values construct a tree representation of the requisite term by joining their operands to a new root node, where only the operation in the root node depends on the specific abstract instruction. The approach is illustrated in Figure 5, 6, 7.

This arrangement makes it easy to extract the terms from program state and convert them to a form appropriate for further processing by the analysis tool. Recall that one of the motivating applications of the proposed approach was symbolic model checking. In this case, the state space is explored by an explicit-state model checker and the extracted terms are converted into SMT queries. To this end, the model checker must be slightly extended and coupled to an SMT solver, since:

1. transitions of the program must be checked for *feasibility*,
2. the state equality check must compare terms semantically, not syntactically.

Of course, the hitherto extracted terms must be left out of byte-wise comparison that is performed on the remaining (concrete) parts of program states. In our case, the required changes in the model checker were quite minor, amounting to about 1200 lines of code.

```
x : a_int ← lift(*)
for i : int ← 1 .. 2
  x : a_int ← a_add(x, 1)
```

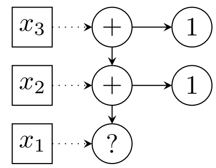

Fig. 7. An example of a formula tree arising from a `for` loop. Versions of the variable `x` which exist in different iterations of the loop are distinguished by an index in the picture.

5.4 Interfaces

One of the goals of the proposed approach was to minimise interfaces between the abstracted program and the verification or execution environment (recall goal 2 set in Section 1.1). In total, there are four interactions at play:

1. non-deterministic choice: under abstraction, conditionals in the program may be undetermined, and both branches may need to be explored; the abstraction uses a non-deterministic choice operator to capture this effect and defers an exploration strategy to the verifier
2. *freeze* and *thaw* must be provided as an interface for storing abstract values in program memory
3. enumeration of enabled (feasible) transitions must take the abstract values into account, if required by the domain(s) used in the program
4. state equality (if applicable in the verification approach) must be extended to take the employed abstract domains into account

The latter two points depend on the chosen abstract domains. For the term domains, both interfaces reduce to extracting abstract values (terms) from program state and executing an SMT query.

6 Evaluation

First of all, we have checked the performance of the transformation itself. On C programs from the SV-COMP suite, the transformation time was negligible. On more complex C++ programs, it took at most a few seconds, which is still fast compared to subsequent analysis.

As described in Section 5, we have built a simple tool which integrates the proposed transformation with an explicit-state model checker and an SMT solver. The experimental evaluation was done using this prototype integration (denoted 'DIVINE*' in summary tables).

6.1 Code Complexity

One of our criteria for the approach presented in this paper was reduced code complexity. While counting lines of code is not a very sophisticated metric, it is a reasonably good proxy for complexity and is easily obtained.[14] The results are summarised in Table 1.

Table 1. Summary of component sizes (thousands of lines of code) in a few symbolic verification and symbolic execution tools. Numbers in parentheses represent shared code (i.e. code not specific to the given approach to symbolic computation).

component	DIVINE*	KLEE	SymDIVINE	CBMC
transformation	3.2	0	0	(22)
virtual machine	(10)	15	6	7.5
exploration	(1.5)	1.2	1	2.3
solver integration	1.2	8	0	14
SAT solver	(45)	(45)	(23)	(5.5)
SMT solver	(80)	(80)	(400)	16
runtime support	1	0	0	0
total unique	5.4	24.2	7	39.8
total shared	136.5	125	423	27.5

6.2 Benchmarks

For benchmarking, we have used a subset of the SV-COMP [6] test cases, namely 7 categories, summarised in Table 2, along with statistics from our prototype tool. We have only taken examples with finite state spaces since the simple approach outlined in Section 5.3 cannot handle infinite recursion or infinite accumulation loops. In total, we have selected 1160 SV-COMP inputs. In many cases (especially in the `array` category), the benchmarks are parametric: we have included both the original SV-COMP instance and smaller instances to check that the approach works correctly, even if it takes a long time or exceeds the memory limit on the instances included in SV-COMP.

In all cases, the time limit, for each test case separately, was 10 minutes (wall time) and the memory limit was 10 GiB. The test machines were equipped with 4 Intel Xeon 5130 cores clocked at 2 GHz and 16 GiB of RAM.

In addition to the present approach, we have measured two additional tools: CBMC 5.8 and SymDIVINE, both of which are symbolic model checkers targeting C code. The overall results of the comparison, in terms of the number of cases solved, are presented in Table 3.

6.3 Comparison 1: CBMC

The results from CBMC 5.8 were obtained using the tool's default configuration. CBMC [20] is a mature bounded model checker for C programs with a good

[14] We have used the utility `sloccount` to get estimates of module size in terms of lines of code.

track record in SV-COMP and is built around a symbolic interpreter for 'goto programs', its own intermediate form, not entirely dissimilar to CIL or LLVM in its spirit. Besides KLEE, the CBMC toolkit is among the best established members of the interpretation-based school of symbolic computation.

Table 2. Summary of test cases from SV-COMP. The time limit was 10 minutes and memory limit was 10 GiB. The oot/oom column is the number of test cases that were executed, but did not finish within the given resource constraints. The 'states' column gives the number of states stored, 'search' gives the state space exploration time and 'ce' gives the counterexample generation time.

tag	solved	oot/oom	states	search	ce
array	96	94	170.3 k	52:00	54:15
bitvector	17	15	3166	3:12	2:33
loops	72	106	14.0 k	53:52	11:40
product-lines	336	239	20.2 M	4:36:44	43:11
pthread	9	36	609.4 k	3:31	0:54
recursion	47	34	3955	16:16	7:41
systemc	14	45	25.0 k	3:29	1:34
total	591	569			

Table 3. The number of benchmarks correctly solved by each of the evaluated tools. The best result in each category is rendered in boldface.

tag	total	DIVINE*	SymDIVINE	CBMC
array	190	**96**	68	93
bitvector	32	**17**	9	2
loops	178	**72**	67	9
product-lines	575	336	**411**	234
pthread	45	**9**	0	1
recursion	81	**47**	43	22
systemc	59	14	**27**	0
total	1160	591	**625**	361

Besides the total number of test cases solved (within the 10 minute limit), we were interested in comparing the time required to do so. Time requirements are summarised in Table 4.

With regards to its state space exploration strategy, CBMC can be thought of as the middle ground between the approach taken by KLEE and that of our proposed tool. On one hand, KLEE, being a symbolic executor, does not attempt to identify already-visited program states. CBMC is a bounded model checker, which means it stores a single formula representing the entire set of reachable states. Our present approach, being based on an explicit-state model checker, stores sets of program states and compares them for equality using an SMT solver.

Table 4. Speed comparison: the columns 'models$_1$' and 'models$_2$' show the number of models which the respective pair of tools finished in common. In most cases, CBMC is substantially faster than the proposed approach, while SymDIVINE is substantially slower.

tag	models$_1$	DIVINE*	CBMC	models$_2$	DIVINE*	SymDIVINE
array	73	34:16	13:58	58	3:18	42:54
bitvector	2	0:37	0:01	9	0:55	2:30
loops	4	0:03	0:02	62	22:25	19:04
product-lin.	182	4:08:24	7:25	183	0:30	28:33
pthread	0	0	0	0	0	0
recursion	22	0:01	0:13	43	4:02	13:58
systemc	0	0	0	14	3:29	6:43

6.4 Comparison 2: SymDIVINE

SymDIVINE [23] is a pre-existing, interpretation-based symbolic model checker which also works with LLVM bitcode. Similar to our approach, SymDIVINE relies on a state equality checker, in this case based on quantified bitvector formulae. In theory, this yields coarser state equivalence and consequently smaller state spaces, but we could not confirm this in our set of benchmarks: the total number of states stored across the benchmarks that finished using both tools was 802 thousand for SymDIVINE and 93 thousand with the approach described in this paper. Additionally, QBV satisfiability queries are typically much more expensive than those used by our prototype tool, which can help explain the speed difference between the tools.

7 Conclusion

We have presented an alternate approach to symbolic execution (and abstract interpretation in general), based on compilation-based techniques, instead of relying on the more traditional interpreter-based approach. We have shown that the proposed approach has important advantages and no serious drawbacks. Most importantly, our technique is modular to a degree not possible with symbolic or abstract interpreters. This makes implementation of software analysis and verification tools based on symbolic execution almost trivial. An important side benefit is that the approach allows for abstract domains other than the term domain, leading to a different class of verification algorithms with a comparatively small investment.

Bibliography

[1] *Principles of program analysis*, 1999. Springer. ISBN 978-3-540-65410-0. doi: 10.1007/978-3-662-03811-6.

[2] Alfred V. Aho. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley series in computer science. Pearson/Addison Wesley, 2007. ISBN 9780321486813.

[3] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From Under-Approximations to Over-Approximations and Back. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *LNCS*, pages 157–172, Berlin, Heidelberg, 2012. Springer.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[5] Petr Bauch, Vojtech Havel, and Jiri Barnat. Control explicit-data symbolic model checking. *ACM Trans. Softw. Eng. Methodol.*, 25(2):15:1–15:48, 2016. doi: 10.1145/2888393.

[6] Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses report on SV-COMP 2016. In *TACAS*, pages 887–904. Springer, 2016. ISBN 978-3-662-49673-2. doi: 10.1007/978-3-662-49674-9_55.

[7] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, volume 6806 of *LNCS*, pages 184–190. Springer, 2011. ISBN 978-3-642-22110-1.

[8] Dirk Beyer and Stefan Löwe. Interpolation for Value Analysis. In *Software Engineering & Management*, volume 239 of *LNI*, pages 73–74. GI, 2015.

[9] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, Oct 2007. ISSN 1433-2787. doi: 10.1007/s10009-007-0044-z.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2187-9. doi: 10.1109/ASE.2008.69.

[11] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[12] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

[13] Marek Chalupa, Martina Vitovská, Martin Jonáš, Jiří Slabý, and Jan Strejček. Symbiotic 4: Beyond reachability. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10206 of *LNCS*, pages 385–389. Springer, 2017. ISBN 978-3-662-54580-5.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[15] Jakub Daniel and Pavel Parízek. PANDA: Simultaneous Predicate Abstraction and Concrete Execution. In *Hardware and Software: Verification and Testing*, volume 9434 of *LNCS*, pages 87–103. Springer International Publishing, 2015. ISBN 978-3-319-26287-1.

[16] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 519–531. Springer, 2007. ISBN 978-3-540-73368-3.

[17] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[18] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 553–568. Springer-Verlag, 2003.

[19] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. ISSN 0001-0782.

[20] Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *TACAS*, pages 389–391. Springer, 2014. ISBN 978-3-642-54862-8. doi: 10.1007/978-3-642-54862-8_26.

[21] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[22] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. ISBN 0792393805.

[23] Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration. In *Model Checking Software*, volume 9641 of *LNCS*, pages 208–213. Springer, 2016.

[24] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In R. Nigel Horspool, editor, *Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2002. Springer. ISBN 978-3-540-45937-8.

[25] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *CAV*, pages 419–423, 2006.

[26] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[27] Marcelo Sousa, César Rodríguez, Vijay D'Silva, and Daniel Kroening. Abstract Interpretation with Unfoldings. In *Computer Aided Verification (CAV)*, volume 10427 of *LNCS*, pages 197–216. Springer, 2017.

[28] Georg Weißenbacher. *Program Analysis with Interpolants*. PhD thesis, University of Oxford, 2010.