Accurate Theorem Proving for Program Verification*

Byron Cook[1], Daniel Kroening[2], and Natasha Sharygina[3]

[1] Microsoft Research
[2] ETH Zurich
[3] University of Lugano

Abstract. Symbolic software verification engines such as SLAM and ESC/JAVA often use automatic theorem provers to implement forms of symbolic simulation. The theorem provers that are used, such as SIMPLIFY, usually combine decision procedures for the theories of uninterpreted functions, linear arithmetic, and sometimes bit vectors using techniques proposed by Nelson-Oppen or Shostak. Programming language constructs such as pointers, structures and unions are not directly supported by the provers, and are often encoded imprecisely using axioms and uninterpreted functions.

In this paper we describe a more direct and accurate approach towards providing symbolic infrastructure for program verification engines. We propose the use of a theorem prover called COGENT, which provides better accuracy for ANSI-C expressions with the possibility of nested logic quantifiers. The prover's implementation is based on a machine-level interpretation of expressions into propositional logic. COGENT's translation allows the program verification tools to better reason about finite machine-level variables, bit operations, structures, unions, references, pointers and pointer arithmetic.

This paper also provides experimental evidence that the proposed approach is practical when applied to industrial program verification.

1 Introduction

Program verification engines, such as symbolic model checkers and advanced static checking tools, often employ automatic theorem provers for symbolic reasoning. For example, the static checkers ESC/JAVA [2] and BOOGIE [3] use the SIMPLIFY [4] theorem prover to verify user-supplied invariants. The SLAM [5,6,7,8,9,10] software model-checker uses ZAPATO [11] for symbolic simulation of C programs. The BLAST [12] and MAGIC [13] tools use SIMPLIFY for abstraction, simulation and refinement. Other examples include the INVEST [14] tool, which uses the PVS [15] theorem prover. Further decision procedures used in program verification are CVC-LITE [16], ICS [17] and VERIFUN [18].

The majority of these theorem provers use either the Nelson-Oppen [19] or Shostak [20] combination methods. These methods combine various decision procedures to provide a rich logic for mathematical reasoning.

* This paper is an extended version of [1].

T. Margaria and B. Steffen (Eds.): ISoLA 2004, LNCS 4313, pp. 96–114, 2006.

However, the fit between the program analyzer and the theorem prover is not always ideal. The problem is that the theorem provers are typically geared towards efficiency in the mathematical theories, such as linear arithmetic over the integers. In reality, program analyzers rarely need reasoning for unbounded integers. Linearity can also be too limiting in some cases. Moreover, because linear arithmetic over the integers is not a convex theory (a restriction imposed by Nelson-Oppen and Shostak), the real numbers are often used instead. Program analyzers, however, need reasoning for the reals even less than they do for the integers.

The program analyzers must consider a number of issues that are not easily mapped into the logics supported by the theorem provers. These issues include pointers, pointer arithmetic, structures, unions, and the potential relationship between these features. Additionally, because bit vectors and arrays are not convex theories, many provers do not support them. In those that can, the link between the non-convex decision procedures can be disappointing. As an example, checking equality between a bit-vector and an integer variable is typically not supported.

When using provers such as SIMPLIFY, the program verification tools must encode the features specific to programming languages into the input logic of the theorem prover, and approximate the language semantics with axioms over the symbols used during the encoding. However, using axioms to encode the language semantics has a drawback in that they can interact badly with the heuristics that are often used by provers during axiom-instantiation in order to improve performance—at the expense of accuracy.

Another problem that occurs when using provers such as SIMPLIFY or ZAPATO is that, when a query is not valid, the provers do not supply concrete counterexamples. Some provers provide partial information on counterexamples. However, in program verification this information rarely leads to concrete valuations to the variables in a program, which is what a programmer most wants when a program verification tool reports a potential bug in their source code.

This paper addresses the following question: *When analyzing programs, can we abandon the Nelson-Oppen/Shostak combination framework in favor of a prover that performs a basic and precise translation of program expressions into propositional logic?*

Inspired by the success of CBMC [21] and UCLID [22], this paper describes a new theorem prover called COGENT which provides direct support for queries in the form of pure ANSI-C [23] expressions together with quantifiers. COGENT largely dispenses with the mathematical theories for unbounded integers and real numbers, and the communication between theories through equivalence relations. Instead, COGENT provides machine-level accurate reasoning for the class of expressions that occur in programs and program invariants.

Much like CBMC, the implementation of COGENT is based on a direct compilation of expressions into propositional logic. When necessary (for example, in order to handle arrays with unbounded size), COGENT uses uninterpreted

functions with Ackerman's encoding. A similar approach is found in UCLID. Pointers are represented as *regions* with finite vectors and offsets.

Cogent's translation allows the program verification tools to accurately reason about arithmetic overflow, bit operations, structures, unions, pointers and pointer arithmetic. Cogent can be used for different software verification applications. As an example, when applied to software model checking it can be used within the abstraction refinement framework [24,25] for abstraction, simulation, and abstraction refinement. Cogent also produces concrete counterexamples to failed proofs.

This paper makes the following novel contributions:

- We provide details on an accurate translation from C expressions together with nested quantifiers into propositional logic. While Cogent is based on parts of the CBMC source code, this paper extends it by using nondeterminism to model architecture dependent behavior. When combined with predicate abstraction, like in SLAM, this technique guarantees that a positive verification result is valid on all standard compliant architectures.
- We demonstrate that the new approach improves the performance of software model checking. In particular, we report results of replacing Slam's theorem prover Zapato with Cogent. This allows us to speed up the verification of previously checked safety properties of Windows device drivers. The speedup is caused by the improved accuracy of Cogent. Moreover, the Cogent-based model checker allows us to verify new properties that make use of bit-level constructs. In this paper, we describe a new Windows device driver bug that was found due to Cogent's improved accuracy. The Zapato-based Slam is unable to locate this bug.
- We also report the results of experiments from queries that come from extended static checking with Boogie.

The queries from Slam and Boogie differ significantly in their characteristics, which allows us to evaluate Cogent's performance under different circumstances. Slam's queries have no quantifiers but make extensive use of structures, pointers, arithmetic and bit operations. Boogie's queries, on the other hand, have nested quantifiers and some uninterpreted functions, but do not use pointer semantics.

The remainder of this paper is organized as follows: Section 2 surveys related work; Section 3 describes the algorithm used by Cogent; Section 4 presents the results of our experiments with Cogent and Slam on benchmarks from Windows device drivers; Section 5 describes the results of Cogent when used to verify conditions generated by Boogie. Section 6 concludes the paper and Section 7 discusses future work.

2 Related Work

In this work we are following the basic proof strategy used by CBMC [21] and UCLID [22]. 1) The input logic of Cogent is translated eagerly into proposi-

tional logic. 2) The resulting propositional formula is then passed to an efficient SAT solver.

The difference between our approach and UCLID is the logic supported by the provers. UCLID does not support the low-level programming language features that CBMC does. On the other hand, CBMC does not support features such as λ-abstraction, which is supported by UCLID.

The experimental application of UCLID to software verification is limited to a restricted set of theorem proving queries from software model checking in [26]. However, neither the relative effect on accuracy nor the effect on the model checking performance was measured, as UCLID was not integrated into an abstraction refinement loop.

To the best of our knowledge (beyond the experiments in [26]), no-one has evaluated the performance of an eager and purely SAT-based theorem prover implementation in abstraction-based symbolic software model checking nor extended static checking. The use of SAT for the abstraction of ANSI-C programs was suggested in [27,28]. No comparative evaluation was done, however, and no support for quantifiers was provided.

CBMC is not unique in its support for accurate reasoning for bit-vectors. Numerous tools implement bit-vector reasoning, particularly hardware verification tools (e.g., [29,30]). Some bit-vector level decision procedures have been adapted to fit into the Nelson-Oppen/Shostak's cooperating decision procedure framework (e.g., [31]). The key difference between the bit-vector support found in CBMC and these provers is that our translation fully accounts for the semantics of the ANSI-C standard [23], using non-determinism in cases where the standard does not specify the details of the machine representation of the data types.

Some program verification tools do not use general purpose theorem provers at all. For example, PREfix [32] and ESP [33] use custom symbolic simulators in which they mix their own language semantics together with the abstractions used in order to make their verification engines scale to large programs. CMC [34] uses a similar approach. Our work is motivated by these efforts. We aim to provide accurate support for the C semantics at the same level of detail as PREfix. Note that CBMC does not provide any abstractions—we expect that the program verification tool performs the abstraction, if needed, while using CBMC for symbolic reasoning.

CBMC builds on the source code of CBMC [21]. CBMC and CBMC differ in that CBMC supports quantifiers and uses non-determinism to take architecture-dependencies into account. They also differ in their intended use: CBMC is designed to be a sound theorem prover for use in any program verification engine, whereas CBMC is a program verification engine by itself.

While not related directly, this work can contribute to the predicate abstraction refinement framework with predicates that contain quantifiers, such as described in [35]. The applications proposed by the authors (hardware and software) would benefit from the accuracy provided in CBMC.

3 Encoding into Propositional Logic

In hardware verification, the encoding of arithmetic operators such as shifting, addition, and even multiplication into propositional logic using arithmetic circuit descriptions is a standard technique. We propose using this same style of encoding in Cogent. This allows us to model artifacts such as arithmetic overflow accurately.

The goal is to implement the ANSI-C standard semantics, as described in [23]. The standard purposely does not provide precise semantics. This is to allow an efficient implementation on different architectures. As an example, the behavior in the case of arithmetic overflow on signed integer types is undefined. Thus, using a true machine-like bit-encoding would be an *under-approximation* of the behavior allowed by the standard. Potentially, this can lead to incorrect verification results, making the verification tool unsound. Therefore, the answers of the prover would be only valid for architectures that use the same bit-encoding. On other architectures, the program might execute in a different way.

In order to avert this problem, we model the architecture-dependent parts of the language semantics by introducing non-determinism into the encoding. A non-deterministic choice can be encoded in propositional logic by using free, unconstrained variables. In order to decide whether to use the non-deterministic choice or not, we add additional checks to the arithmetic operators. If an operator obtains operands for which the result is architecture dependent, the result of the operator is a non-deterministic choice.

In the context of software verification, if the prover reports that the property is verified, the property holds for any architecture compliant with the standard.

3.1 Scalar Data Types

The scalar data types are encoded using a particular bit width for each data type. This bit-width is a run-time option. The arithmetic operators (e.g., addition, multiplication, division) and the bit-wise operators are transformed into corresponding arithmetic circuits using basic gates such as AND, OR, NOT. These circuits are then transformed into propositional logic.

Optimizations for Division. While a standard arithmetic circuit for addition, subtraction, multiplication and shifting provides sufficient performance, implementing an iterative division circuit using propositional logic is prohibitively expensive. We therefore implement the division and remainder operators as follows: we use non-deterministic choice to *guess* the correct result of the division, i.e., the quotient q and the remainder r, and then add constraints that these guesses are correct. I.e., we return q, r such that $q * b + r = a$. This requires one multiplication, one addition and one equality test. Note however, that the multiplication and the addition must be forced (by adding appropriate constraints) not to overflow, or wrong results would be obtained.

Arithmetic Overflow on Unsigned Types. On unsigned integer types, the ANSI-C standard requires modular arithmetic, i.e., the result is required to be a bit-encoding of $r \bmod 2^n$, where r is the result obtained with infinite precision and n is the number of bits. Using arithmetic circuits accurately models these semantics, so no non-determinism is required.

Arithmetic Overflow on Signed Types. On signed integer types, the ANSI-C standard leaves the behavior in case of arithmetic overflow undefined. In particular, the semantics of a two's complement encoding are not guaranteed.

Formally, let $overflow+(a, b)$ denote a Boolean function that is true if and only if the sum of a and b is outside the interval given by `INT_MIN` and `INT_MAX`. Let $a \oplus b$ denote the bit-vector operator for adding a and b. Let $\perp$ denote a vector of free, new variables with the same width as the addition result. The result of a signed addition is denoted by op_s+.

$$op\_s+(a, b) := overflow+(a, b)?\perp : a \oplus b$$

Note that the case-split on the overflow is translated as part of the circuit, and thus performed dynamically by the propositional logic solver, not during translation. A similar definition is used for subtraction, multiplication, and bitwise shifting.

3.2 Structures, Unions, and Bounded Arrays

Structures and small arrays are encoded in a straight-forward manner by recursively concatenating the bit-vectors that encode their components. Large arrays are treated like arrays with unbounded size, which is described in the next section. The prover query language contains operators to extract members from a structure and to replace members. If an array index operation is out of bounds, the value of the index operator is a vector of free variables, i.e., it is non-deterministically chosen. In contrast to that, when using a conventional theorem prover such as Simplify, arrays and structures are typically encoded using uninterpreted functions and axioms. This requires expensive heuristics for quantifier instantiation.

Unions are encoded using a pair of bit-vectors. The first bit-vector is as wide as the widest bit-vector of any of the union members. It encodes the value of the union. The second bit-vector is a binary encoding of the number of the member that was used last for writing into the union. During member extraction, we check that the extracted member matches the member used for writing. If they do not match, the value of the member extraction operator is a vector of free variables.

3.3 Unbounded Arrays

Programs may allocate arrays of variable size. Encoding such an array using a bit-vector is infeasible. Thus, we model unbounded arrays as uninterpreted functions using Ackermann's reduction, as done in CBMC [21]. Note that the contents of the array are still interpreted as bit-vectors.

3.4 Pointers

We encode the value of a pointer using two bit-vectors. Let p denote a pointer type expression. The first bit-vector, denoted by $p.o$, encodes the object the pointer points to, while the second bit-vector, denoted by $p.i$, encodes an index within that object using two's complement. The width of $p.i$ is the same as the width used for the integer type. The width of $p.o$ is dynamically adjusted to accommodate the number of objects mentioned within the query. The object bit-vector consisting of all zeros is used to encode a `NULL` pointer[1].

The offset bit-vector is used to encode the position of the pointer within the object. In case of an array consisting of elements of a scalar data type, this value is equal to the array index, independent of the size of the scalar data type. Structures consisting of n fields with scalar data types are treated like an array with n elements, even if the types of the individual fields have different widths.

If arrays and structures are nested, the offset bit-vector is equal to the number of the scalar type variable inside the nested data structure.

Address Operator and Pointer Arithmetic. The encoding above models the semantics of the ANSI-C pointer operators accurately. The unary `&` operator returns the address of the object passed as operand. The operand may contain field access and array index operators. These operators are handled by adjusting the index bit-vector. As the array index may be a variable, the formula built for the index bit-vector may require addition and multiplication.

The pointer arithmetic operands only adjust the index bit-vector, never the object bit-vector. The logic includes predicates that allow checking for overflow and underflow on pointer arithmetic operations, if desired.

Function Pointers. Functions mentioned in the query are assigned object numbers just as variables. However, the ANSI-C standard provides no semantics for arithmetic on pointers pointing to functions. Thus, the pointer arithmetic operators return non-deterministic results when applied to pointers that point to functions.

Relational Operators. When checking equality between two pointers, as specified by the ANSI-C standard, the object bounds have to be considered. If the index bit-vector of the pointer is not within the object, we call the pointer out-of-bounds. As a special case, the index bit-vector of the pointer can be exactly one element beyond the end of the object. We call such a pointer an off-by-one pointer. In case of a pointer pointing to the `NULL` object, any index bit-vector other than zero is considered to be out of bounds. These comparisons are done dynamically by encoding an arithmetic circuit for the relations on the index bit-vector and the object size, which may be a variable.

We form an equation that dynamically distinguishes the following cases:

- If both pointers are within their bounds, the result of the comparison is equal to bitwise equality of both components of the pointer.

[1] Note that the ANSI-C standard prohibits dereferencing a `NULL` pointer. It is a common misunderstanding that dereferencing `NULL` will result in the value zero.

- If both pointers point to the same object (i.e., the object bit-vectors are bitwise equal) and both pointers are within their bounds or off-by-one, the result of the comparison is equal to bitwise equality of the index bit-vector.
- Otherwise, the result is a free, unconstrained variable.

When checking the other relations (greater than, and so on), the standard requires that the two pointers must point to the same object. Also, the pointer must be within the bounds or off-by-one. The result of the comparison is a non-deterministic choice if either check fails.

3.5 Quantifiers

The sections above describe the translation of formulae into propositional logic. In these formulae, all variables are assumed to be implicitly universally quantified. However, some program analysis tools make use of nested quantifiers. In most cases, CODGENT is able to rewrite the input query in such a way that the quantifiers can be encoded directly into propositional logic with fresh variables and Skolemization. In the worst case, CODGENT will translate the input formula into propositional logic with quantifiers (called *quantified Boolean formulae* or QBF) instead of the standard propositional logic—we believe that this case will not occur frequently in practice.

3.6 Examples

In order to summarize the techniques above, consider the following examples. Given the formula Q

```
p+x!=q || &(p->y) == &((q-x)->y)
```

let `p` and `q` be two pointers to a structure containing a member `y`. Let `x` be an integer variable.

This formula is translated into propositional logic as follows. First, two new Boolean variables α and β are allocated for the two operands of the OR operator. Then, we add the following constraint:

$$Q \iff \alpha \vee \beta$$

We then add the constraints for the left-hand side operand of the OR operator. We allocate bit-vectors for $p.i$, $p.o$, $q.i$, $q.o$, and x. We assume that n is the number of elements of simple type in the structure, and that $\otimes$ is the bit-vector multiplication operator.

$$\alpha \iff (p.i \oplus (x \otimes n) \neq q.i) \vee (p.o \neq q.o)$$

Note that this constraint does not contain the bounds check for the object pointed to by `p` and `q`.

For the encoding of the right-hand side of the OR operator, suppose that `y` is the second member of the structure. Thus, the index bit-vector is increased by one when taking the address of `p->y`.

$$\beta \iff (p.i \oplus 1 = (q.i \ominus (x \otimes n)) \oplus 1) \wedge (p.o = q.o)$$

This simple example illustrates the complexity of mixing pointer arithmetic with structures and arrays. In contrast to our tool, existing decision procedures are unable to handle even such simple examples.

In order to illustrate an invalid query generating a counterexample, consider the formula R

```
!(p==a+2 && q==a+n && p==q)
```

where `a` is an array, `p` and `q` are pointers, and `n` is an integer. Again, we first assign fresh Boolean variables α_2 β_2, and γ_2 to the operands of the AND operator:

$$R \iff !(\alpha_2 \wedge \beta_2 \wedge \gamma_2)$$

The encoding of the constraints for the pointer arithmetic is done similarily as above for α. The object `a` is assigned a number. Suppose this number is 1. When passed to a SAT solver, we obtain a satisfying assignment with `n` = 2, a value of 1 for the object of `p` and `q`, and a value of 2 for the offset of `p` and `q`.

4 Application to Software Model Checking

One popular approach to software model checking is called *counter-example guided abstraction refinement* (CEGAR). SLAM, for example, implements CEGAR for the C programming language. CEGAR implementations [24,25,36,37,13, 14,12] often use automatic theorem provers to implement the abstraction and refinement components of this algorithm. In this section, we briefly describe the CEGAR approach, and then present results of an experiment with SLAM where we have replaced the theorem prover ZAPATO with COGENT.

4.1 Software Model Checking with Counter-Example Guided Abstraction-Refinement

Predicate abstraction [38,39] is a method for systematically constructing conservative abstractions of software. It abstracts data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The predicate abstraction of software is usually automated. For example, in SLAM, the predicate abstraction is implemented in a module called C2BP [40,5].

In practice, the set of predicates must be discovered by trial-and-error. Typically, CEGAR implementations guess the initial set of predicates. If the abstraction is computed using an insufficient set of predicates, then the model checker will find a false error in the abstraction—called a *spurious trace*. There are two

sources of spurious traces: 1) the set of predicates is insufficient, and 2) C2BP trades accuracy for efficiency.

SLAM first uses NEWTON [6] to symbolically simulate the entire trace and determine if it is spurious. If the trace is spurious, then NEWTON searches for additional predicates which could eliminate the trace in a refined abstraction.

If no new predicates are found, SLAM concludes that the spurious trace is caused by the inexact abstraction done by C2BP. It then invokes another refinement method, called CONSTRAIN [7]. CONSTRAIN symbolically examines each step of the trace in isolation and attempts to refine the abstract transition relation in order to improve the accuracy of the abstraction using the predicates that are available.

By default, NEWTON and CONSTRAIN both use the theorem prover ZAPATO [11]. As is done in [26], C2BP does not call a theorem prover. Instead, it uses a module called FASTCOVERING that implements a form of parallel inference.

4.2 Experiments with SLAM

We have integrated COGENT with SLAM and compared the results to SLAM using its original theorem prover, ZAPATO. Note that, in our integration, C2BP still uses FASTCOVERING. FASTCOVERING is currently an extremely weak inference engine that produces poor abstractions when uncommon symbols (like the C bitwise operations) appear in the sets of predicates. As a consequence, SLAM/COGENT is at a disadvantage over SLAM/ZAPATO, as the abstraction of bitwise operations must be done in a needlessly inefficient manner with CONSTRAIN. For a more optimal result, FASTCOVERING should perform an analysis similar to COGENT in order to provide better abstractions.

4.2.1 Comparing the Model Checking Results

In order to compare the overall effect of COGENT on SLAM we ran SLAM/COGENT on 308 model checking benchmarks and compared the results to SLAM/ZAPATO. The results are given in Fig. 1.

Model checking result	SLAM/ZAPATO	SLAM/COGENT
Property passes	243	264
Time threshold exceeded	39	17
Property violations found	17	19
Cases of abstraction-refinement failure	9	8

Fig. 1. Comparison of SLAM/ZAPATO to SLAM/COGENT on 308 device driver correctness model checking benchmarks. The time threshold was set to 1200 seconds.

The SLAM/COGENT performs considerably better that SLAM/ZAPATO. Notably, the number of cases where SLAM exceeded the 1200 second time threshold was reduced by half. As a result, the reduced timeouts led to two additional bugs

being found. The cases where SLAM failed to refine the abstraction (as described in detail in [7]) was effectively unchanged.

During SLAM's execution, the provers actually returned different results in some cases. This is expected, as the provers support different logics:

- ZAPATO provides support for uninterpreted functions together with UTVPI integer arithmetic [41]. In addition, ZAPATO supports expressions with pointers only through axioms and a heuristic for dynamic axiom instantiation.
- COGENT, on the other hand, supports full arithmetic over bit vectors together with a more accurate handling of pointers and structures. COGENT is strictly more accurate than ZAPATO.

For this reason, there are queries that ZAPATO can prove valid and COGENT can prove invalid (e.g., when overflow is ignored by ZAPATO), and vice-versa (e.g., when validity is dependent on pointer arithmetic or non-linear uses of multiplication). Thus, it is difficult to compare the accuracy of ZAPATO to COGENT. We have, however, compared the overall performance of the two provers and found that COGENT is usually more than 2x slower than ZAPATO. On 2000 theorem proving queries ZAPATO executed for 208s, whereas COGENT ran for 522s. We can therefore conclude that the performance improvement in Fig. 1 is indicative that, while COGENT is slower, COGENT's increased accuracy allows SLAM to do less work overall.

4.3 Checking New Properties of Windows Drivers

During the formalization of the kernel API usage properties that SLAM is used to verify [25], a large set of properties were removed or not actively pursued due to inaccuracies in SLAM's theorem prover. For this reason the results in Fig. 1 are not fully representative of the improvement in accuracy that SLAM/COGENT can give.

In order to demonstrate this improved accuracy, we developed and checked several new safety properties that would have resulted in too many false bugs being reported in SLAM/ZAPATO. Fig. 2 contains an example of such a property, written in SLAM's event-based property language called SLIC [42]. It makes use of COGENT's treatment of bit vectors, structures and pointers. This rule checks that a Windows device driver always sets a special bit in a field of a structure to `0` before returning from its `AddDevice` callback routine.

This property has the effect of instrumenting three events into the driver when SLAM performs symbolic model checking:

- Calls from the device driver to the kernel function `IoCreateDevice`, which (in the case the function returns successfully) causes an assignment of `1` to the variable `created`.
- Calls from the device driver to the kernel function `IoDeleteDevice`, which causes an assignment of `0` to the variable `created`.

```
// The variable "created" is 0 when the special variable pdevobj is not
// pointing to something that has been allocated.  It is set to
// 1 when it is.
state
{
    int created = 0;
}

// IoCreateDevice will, if successful, place the pointer pdevobj in the
// handle passed to it.
IoCreateDevice.exit
{
    if ($return==STATUS_SUCCESS) {
        created = 1;
    }
}

IoDeleteDevice.exit
{
    created = 0;
}

// If the driver has an AddDevice callback, it will be called fun_AddDevice
#ifdef fun_AddDevice
fun_AddDevice.exit
{
    // pdevobj is the pointer returned the environment model
    // for IoCreateDevice
    if (created && (pdevobj->Flags & DO_DEVICE_INITIALIZING) != 0) {
        abort "AddDevice routine failed to set ~DO_DEVICE_INITIALIZING flag";
    }
}
#endif
```

Fig. 2. SLIC device driver safety property using C bit operations

– Returns from the device driver's `AddDevice` callback routine.[2] When this event occurs, a check (under the condition that the device object has been allocated) ensures that the driver has negated the `DO_DEVICE_INITIALIZING` flag in the device object structure that was allocated.

This rule is checked together with a `main` function that calls the driver's `AddDevice` routine from an unspecified state, and a set of non-deterministically abstracted models of the kernel functions that the driver might call.

Fig. 3 displays the environment model for the function `IoCreateDevice` that is used while checking device drivers with SLAM. This function can return any

[2] `AddDevice` is referred to as a C macro called `fun_AddDevice` in the property. Before SLAM is used to perform model checking, an initial scan of the driver's source code is done and special callbacks found during this pass are defined using the C macro language. These macros are then called from the properties and kernel environment model.

```
NTSTATUS
IoCreateDevice(
    DRIVER_OBJECT * DriverObject,
    unsigned long int DeviceExtensionSize,
    UNICODE_STRING * DeviceName,
    DEVICE_TYPE DeviceType,
    unsigned long int DeviceCharacteristics,
    unsigned int Exclusive,
    DEVICE_OBJECT **DeviceObject
    )
{
    switch (MakeNondeterministicChoice()) {
      case 0:  (*DeviceObject) = pdevobj;
               pdevobj->Flags |= DO_DEVICE_INITIALIZING;
               return STATUS_SUCCESS;
      case 1:  (*DeviceObject) = NULL;
               return STATUS_INSUFFICIENT_RESOURCES;
      case 2:  (*DeviceObject) = NULL;
               return STATUS_OBJECT_NAME_EXISTS;
      default: (*DeviceObject) = NULL;
               return STATUS_OBJECT_NAME_COLLISION;
    }
}
```

Fig. 3. Nondeterministic environment model of Windows kernel function `IoCreateDevice` for device driver verification with SLAM

of four possible return values. In the case that it returns `STATUS_SUCCESS` it sets the `DO_DEVICE_INITIALIZING` flag in `pdevobj`'s `Flags` field to `1`.

We checked this new property on 15 Windows device drivers using both SLAM/ZAPATO and SLAM/COGENT. When using ZAPATO, SLAM found false errors in each driver. When using COGENT as the prover, SLAM was able to verify the correctness of all but one driver. In the case of this one driver, SLAM produced a counterexample that pointed to a real and previously unseen bug.

5 Application to Extended Static Checking

BOOGIE [3] is an implementation of Detlef *et al.*'s notion of *extended static checking* [43] for the C# programming language. Extended static checkers attempt to automatically verify manually added pre- and post-conditions in code. It can also be used to ensure that client-code respects the pre-conditions, and does not assume too much of the post-conditions. BOOGIE, using a notion of *weakest-preconditions*, computes *verification conditions* that can be checked by an automatic theorem prover. BOOGIE uses SIMPLIFY to formally validate the conditions.

In order to demonstrate the applicability of COGENT to extended static checking, we have applied it to verification conditions generated by BOOGIE and compared the results to those of SIMPLIFY. The runtimes in seconds are given in Fig. 4.

Benchmark #	COGENT	SIMPLIFY	Benchmark #	COGENT	SIMPLIFY
1	0.010s	0.029s	28	0.044s	0.050s
2	0.013s	0.029s	29	0.030s	0.045s
3	0.012s	0.028s	30	0.002s	0.025s
4	0.041s	0.042s	31	0.003s	0.026s
5	0.573s	0.452s	32	0.093s	0.111s
6	0.001s	0.026s	33	27.772s	16.311s
7	0.002s	0.026s	34	0.001s	0.024s
8	0.002s	0.027s	35	0.001s	0.025s
9	0.042s	0.045s	36	0.002s	0.025s
10	0.043s	0.051s	37	0.010s	0.038s
11	0.030s	0.045s	38	0.014s	0.030s
12	0.002s	0.025s	39	0.012s	0.029s
13	0.003s	0.026s	40	0.042s	0.042s
14	0.093s	0.100s	41	0.573s	0.457s
15	26.217s	15.735s	42	0.002s	0.026s
16	0.001s	0.024s	43	0.001s	0.025s
17	0.001s	0.025s	44	0.002s	0.027s
18	0.002s	0.026s	45	0.042s	0.045s
19	0.010s	0.030s	46	0.043s	0.050s
20	0.013s	0.029s	47	0.030s	0.045s
21	0.013s	0.028s	48	0.002s	0.025s
22	0.042s	0.043s	49	0.003s	0.026s
23	0.571s	0.455s	50	0.092s	0.112s
24	0.001s	0.026s	51	30.813s	70.763s
25	0.001s	0.026s	52	0.001s	0.024s
26	0.003s	0.027s	53	0.001s	0.024s
27	0.042s	0.045s	54	0.001s	0.025s

Fig. 4. Comparison of SIMPLIFY and COGENT on 54 verification conditions generated by BOOGIE

Unlike we did in the case of SLAM, we have not yet fully integrated BOOGIE and COGENT. For the purpose of this experiment we first annotated the variable names in the input C# programs with their types—BOOGIE currently does not pass any type information down to the theorem prover. We then ran BOOGIE on the programs and collected the verification conditions. We converted the queries from SIMPLIFY's input format into the syntax of COGENT, and removed the axioms and artifacts of the SIMPLIFY-specific encoding. Note that the comparison is not quite fair: SIMPLIFY's execution time includes parsing, whereas parsing and translation is not included in the COGENT execution time.

The verification conditions mix both finite and infinite types together with references. Objects of unbounded types were encoded with uninterpreted functions and axioms. The verification conditions did not contain any pointer arithmetic,

nor C-style unions. They did, however, contain some examples with bit-level operations. In particular, one C# program models a microprocessor (as described in some detail in [44]) and makes heavy use of bit-level programming constructs.

For the harder queries, COGENT was faster in one instance, whereas SIMPLIFY was faster in two. Unlike SLAM, BOOGIE does not use the results of the validity checks during its analysis, so the increased accuracy provided by COGENT does not improve the overall performance of BOOGIE.

Note that C# provides an unsafe extension, in which pointer arithmetic and other C-like features can be used. This is, for example, how C# calls C code. Using the increased accuracy of COGENT for low-level programming features, BOOGIE could potentially analyze mixtures of unsafe and safe code.

6 Conclusion

Automatic theorem provers are often used by program verification engines. However, the logics implemented by these theorem provers are not typically ideal for the program verification domain. In this paper, we have described a new prover that accurately supports the type of reasoning that program verification engines require.

The prover's strategy is to directly encode input queries into propositional logic. This encoding accurately supports bit operations, structures, unions, pointers and pointer arithmetic, and pays particular attention to the sometimes subtle semantics described in the ANSI-C standard. We have detailed the prover's translation of queries into propositional logic. We have also reported experimental results that demonstrate the performance and accuracy improvements of the approach. We make the tool and the bitvector benchmark files used available on the web[3] in order to allow other researchers to reproduce our results.

7 Future Work

As future work, we would like to further extend the prover with features that can be useful for symbolic program verification tools. As an example, the prover should take a query that represents a symbolic state of a program and apply a widening operation such that verification engines based on abstract interpretation [45] could potentially reach a fixpoint. Additionally, we would like to make use of interpolants [46,47] in COGENT.

A number of modern automatic theorem provers, such as CVC-LITE, ZAPATO, ICS and VERIFUN, produce proofs. These proofs can be used in cases to quickly determine *why* a query is valid. When used for symbolic simulation, this allows us to find a small set of facts that cause a trace to be spurious. SAT-solvers for propositional logic typically can produce an *unsatisfiable core* which has similar information. For this reason, COGENT is able to produce information that is similar—but not identical—to the proofs generated by traditional provers. In

[3] `http://www.inf.ethz.ch/personal/kroening/cogent/`

the future we would like to demonstrate that the unsatisfiable cores can provide the same benefit to symbolic simulators as the proofs.

As mentioned in Section 4, the abstraction module of SLAM uses FASTCOVERING, which is similar to [26] but optimized for speed and not precision. The motivation behind this approach is to avoid the exponential number of calls to a theorem prover—as originally proposed in [38]. As we replaced ZAPATO with COGENT in Section 4, we would also like to replace FASTCOVERING with a new module that supports the same level of accuracy as COGENT.

There are a number of areas for potential performance improvement in COGENT. We would like to optimize COGENT and then perform more extensive empirical comparisons. Additionally, we would like to better integrate COGENT with BOOGIE and compare the approaches on a larger set of benchmarks.

Acknowledgments

The authors would like to thank Mike Barnett, Sergey Berezin, Vijay Ganesh, Rustan Leino, Madan Musuvathi, and Lintao Zhang for their ideas and comments related to this work.

References

1. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In Etessami, K., Rajamani, S.K., eds.: Proceedings of CAV 2005. Volume 3576 of Lecture Notes in Computer Science., Springer Verlag (2005)
2. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 02: Programming Language Design and Implementation. (2002)
3. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology **3** (2004) 27–56
4. Detlefs, D., Nelson, G., , Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
5. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 01: Programming Language Design and Implementation, ACM (2001) 203–213
6. Ball, T., Rajamani, S.K.: Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research (2002)
7. Ball, T., Cook, B., Das, S., Rajamani, S.K.: Refining approximations in software predicate abstraction. In: TACAS 04: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2004)
8. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130
9. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN 00: SPIN Workshop. LNCS 1885. Springer-Verlag (2000) 113–130

10. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: PASTE 01: Workshop on Program Analysis for Software Tools and Engineering, ACM (2001) 97–103
11. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic theorem proving for predicate abstraction refinement. In: CAV 04: International Conference on Computer-Aided Verification. (2004)
12. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread modular abstraction refinement. In: CAV 03: International Conference on Computer-Aided Verification, Springer Verlag (2003) 262–274
13. Chaki, S., Clarke, E., Groce, A., Strichman, O.: Predicate abstraction with minimum predicates. In: CHARME 03: Advanced Research Working Conference on Correct Hardware Design and Verification Methods. (2003)
14. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: TACAS 01: Tools and Algorithms for the Construction and Analysis of Systems. (2001)
15. Owre, S., Shankar, N., Rushby, J.: PVS: A prototype verification system. In: CADE 11: International Conference on Automated Deduction. (1992) Saratoga Springs, NY.
16. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity checker. In: CAV 04: International Conference on Computer-Aided Verification. (2004)
17. Filliatre, J.C., Owre, S., Rue, H., Shankar, N.: ICS: Integrated canonizer and solver. In: CAV 01: International Conference on Computer-Aided Verification. (2001)
18. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 355–367
19. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems **1** (1979) 245–257
20. Shostak, R.E.: Deciding combinations of theories. Journal of the ACM **31** (1984) 1–12
21. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems. (2004)
22. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV 02: International Conference on Computer-Aided Verification. (2002)
23. International Organization for Standardization: ISO/IEC 9899:1999: Programming languages — C. International Organization for Standardization, Geneva, Switzerland (1999)
24. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1995)
25. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: IFM 04: Fourth International Conference on Integrated Formal Methods. (2004)
26. Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: CAV 03: International Conference on Computer-Aided Verification. (2003) 141–153
27. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Technical Report CMU-CS-03-186, Carnegie Mellon University, School of Computer Science (2003)

28. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design **25** (2004) 105–127
29. Aagaard, M., Jones, R., Melham, T., O'Leary, J., Seger, C.J.H.: A methodology for large scale hardware verification. In: FMCAD 02: Formal Methods In Computer-Aided Design. (2002)
30. Grundy, J.: Verified optimizations for the Intel IA-64 architecture. In: TPHOLs 00: Theorem Proving in Higher-Order Logics. (2000)
31. Barret, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for bit-vector arithmetic. In: DAC 98: Design Automation Conference. (1998)
32. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. Software—Practice and Experience **30** (2000) 775–802
33. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: PLDI 02: Programming Language Design and Implementation. (2002)
34. Musuvathi, M.S., Park, D., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. In: OSDI 02: Operating Systems Design and Implementation. (2002)
35. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Proc. of the 5th Intl. Conference on Verification, Model Checking and Abstract Interpretation (VMCAI). Number 2937 in LNCS, Springer-Verlag (2004) 267–281
36. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV 00: International Conference on Computer-Aided Verification. (2000)
37. Das, S., Dill, D.L.: Successive approximation of abstract transition relations. In: Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science. (2001) June 2001, Boston, USA.
38. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: CAV 97: Conference on Computer Aided Verification. Volume 1254 of Lecture notes in Computer Science., Springer-Verlag (1997) 72–83 June 1997, Haifa, Israel.
39. Colón, M.A., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: CAV 98: Conference on Computer-Aided Verification. Volume 1427 of Lecture Notes in Computer Science., Springer-Verlag (1998) 293–304
40. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstractions for model checking C programs. In: TACAS 01: Tools and Algorithms for Construction and Analysis of Systems. LNCS 2031, Springer-Verlag (2001) 268–283
41. Harvey, W., Stuckey, P.: A unit two variable per inequality integer constraint solver for constraint logic programming. In: Australian Computer Science Conference (Australian Computer Science Communications). (1997) 102–111
42. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research (2001)
43. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Technical Report 159, Compaq Systems Research Center (1998)
44. Gurevich, Y., Wallace, C.: Specification and verification of the Windows Card runtime environment using abstract state machines. Technical Report MSR-TR-99-07, Microsoft Research (1999)
45. Cousot, P.: Abstract interpretation. Symposium on Models of Programming Languages and Computation, ACM Computing Surveys **28** (1996) 324–328

46. Thomas A. Henzinger, Ranjit Jhala, R.M., McMillan, K.L.: Abstractions from proofs. In: POPL 04: Principles of Programming Languages, ACM Press (2004) 232–244
47. McMillan, K.: An interpolating theorem prover. In: TACAS 04: Tools and Algorithms for Construction and Analysis of Systems, Springer-Verlag (2004)