Formal Verification of a Java Component Using the RESOLVE Framework

Laine Rumreich and Paolo A. G. Sivilotti

The Ohio State University, Columbus OH, USA
rumreich.1@osu.edu, paolo@cse.ohio-state.edu

Abstract. A Binary Decision Diagram (BDD) is an efficient representation of a Boolean formula with many applications in model checking, SAT solving, networking, and artificial intelligence. This paper uses the RESOLVE specification and reasoning framework to formally verify the functional correctness of a Java implementation of a BDD component. RESOLVE uses rich mathematical abstractions and clean value-based semantics for modular reasoning of assertive code. Java, on the other hand, includes many language features that are inconsistent with this notion of clean semantics and modular reasoning. Aliases, in particular, are easily created via assignment, parameter passing, and iterators, so reference-based semantics and points-to analysis are usually necessary when reasoning about Java code. This paper demonstrates the combination of these two paradigms. The implementation uses Java, but in a disciplined way and layered on a component catalog expressly designed to support modular reasoning. The assertional aspects of the code use RESOLVE, but are tailored to Java syntax and language constructs. In the development of the correctness proof for the BDD component, several errors in the original Java implementation were discovered and corrected. These errors were present despite the implementation passing an extensive test suite, exhibiting the value of the proof. The verification also exposed a limitation in the more general component design pattern related to unreachable code.

Keywords: Formal Verification · Value Semantics · Modularity · Binary Decision Diagram.

1 Introduction

In order to be tractable, software verification must be modular. That is, the correctness of the entire system must follow from the correctness of its individual components. Modularity allows for components to be verified in isolation and for the cost of this verification to be amortized over the re-use of these components across multiple systems.

The work presented in this paper uses the RESOLVE (REusable SOftware Language with Verifiability and Efficiency) framework [34], which is both an integrated implementation and verification system as well as a design discipline

aimed to promote component re-use by easing client-side reasoning. The implementation notation uses assertional code with value semantics and the associated verification system uses extendable mathematical theories with custom decision procedures for automated verification. The discipline prescribes principles for interface design, the tenets of which reduce overall software cost and improve quality [40]. While the RESOLVE discipline is typically applied to software components written in the RESOLVE language, elements of the discipline could, in principle, also be adapted for use with industry-standard programming languages such as Java. This combination of Java and RESOLVE would leverage the strengths of both: the robust full-functional verification possible in RESOLVE and the practicality of an industry-standard programming language such as Java. This paper considers the feasibility of such a combination.

The primary contribution of this work concerns the careful construction of a proof that formally establishes the correctness of a Java-based Binary Decision Diagram component, the implementation of which is layered on top of a RESOLVE-style component library with RESOLVE specifications. Because industry-standard languages such as Java and C were not designed with verification in mind, they include many features that challenge the soundness of modular reasoning, including: aliasing-by-default, reference semantics, ubiquitous side-effects, and concurrency. Because of these challenges, the verification of software written in these languages is often restricted to subsets of properties of interest (*e.g.*, race detection) or limited in generality (*e.g.*, reasoning over memory locations rather than abstract mathematical values).

In contrast, the verification effort described in this paper entails both full-functional correctness and value-based reasoning. As a result of this verification, several errors were identified in the original BDD component implementation. In addition, we present several observations related to the success of adapting RESOLVE for application in the context of the Java programming language.

2 Background

2.1 Previous Work

While the RESOLVE discipline is typically applied to programs written in the RESOLVE programming language, there exists a substantial library of RESOLVE-style components written in the Java programming language. None of these components have been formally verified, however, despite their adherence to the discipline. This paper describes the verification of one of these components, `BooleanStructure`, which implements a Binary Decision Diagram (BDD). BDDs are frequently used and versatile data structures that represent Boolean formulas. BDDs have unique features that make them preferable to other common representations of Boolean formulas such as truth tables and propositional formulas. These features improve the efficiency of BDDs and make them more useful than other common representations for solving complex problems with many variables. In 2018, Asim [3, 2] developed a Binary Decision Diagram software component in the Java programming language. Unlike existing BDD components,

this BDD was developed with verification in mind, including behavioral specifications. This component was carefully constructed following the RESOLVE discipline, including the definition of an abstract mathematical model of state, an interface supporting observability and controllability, and a layered implementation that separates core functionality from secondary operations. This software component has been formally verified in this work and is accessible on GitHub [35].

2.2 Existing BDD Implementations

Existing software packages with implementations of the BDD data structure are available for commonly used programming languages such as Java, C, and Python, including BuDDy [25], CuDD [37], CacBDD [28], SableJBDD [31], JDD [38], Sylvan [11], and BeeDeeDee [27]. Of these packages, SableJBDD and JDD are Java-based. None of these packages have been formally verified or include formal specifications, however, so correctness guarantees cannot be made and the formal verification of these components would require substantial effort.

2.3 Applications of BDDs

BDDs are useful data structures in a variety of contexts, including disciplines where correctness is critical. Some examples of these disciplines include Boolean satisfiability, circuit design [24], formal verification, symbolic model checking, network analysis [39], and artificial intelligence [23, 26].

BDDs have commonly been used in formal verification and model checking, which are highly relevant to this work due to their strict correctness requirements. Symbolic model checking based on BDDs was originally used in hardware model checking but was later extended to the domain of software verification. Prior to the introduction of BDD-based symbolic model checking, practical hardware verification via model checking was limited to models with less than 10^6 reachable states [9]. BDDs enabled practical verification of industrial systems with state spaces of more than 10^{20} [8], which also allows for software verification. Since they were popularized, BDDs have been frequently used in formal software verification and symbolic model checking, including in the Berkeley package HSIS for formal verification [4].

Despite the popularity and frequent use of the data structure, existing BDD components have not been formally verified. Thus, guarantees about their correctness cannot be made. This is particularly relevant for formal verification and symbolic model checking applications because the tools being used for verification purposes are not verified themselves.

2.4 Object-Oriented and Automatic Verification

This work concerns the verification of a Java software component consisting of two interfaces, an abstract base class, and a concrete derived class. A substantial

barrier to this verification process and to reasoning about Java programs in general is the presence of aliases. Reasoning about software with aliases is well-known as a challenging problem [18]. Many techniques to control aliasing in object-based languages have been proposed, including notions of ownership and borrowing as in Rust [20], adding additional annotations just for aliasing [1], and the use of separation logic [7]. Another approach is to modify the language to prevent aliases altogether, such as requiring pointers to be unique [29] or using swapping to avoid aliasing [22]. The RESOLVE discipline, which is used in the component verified in this project, uses the notions of swapping and clean semantics to avoid problems related to aliasing [21]. This paper demonstrates some of the additional complexities of reasoning about aliasing in Java rather than the RESOLVE programming language.

Considerable research exists in the area of automatic software verification toward the goal of eliminating programming errors. Despite significant advances in automated theorem provers, SAT solvers, and SMT solvers, the construction of a fully automatic verifying compiler remains a long-term challenge in computer science. Challenges including aliasing, side-effects, fixed-width number representations, and concurrency make verification of object-oriented languages especially challenging. There has been relative success in this area using custom programming languages explicitly designed with verification in mind. Verification engines for such languages have been built using Dafny [15], RESOLVE [33], Why3 [12], and Whiley [30]. These verifying compilers leverage formal reasoning constraints built into the programming language to simplify automatic verification. Some of these languages, such as RESOLVE and Why3, can be translated to other languages such as C, Java, or Ada, but require that the program is first developed in the language designed for formal verification. However, these verifiers cannot be used for the BDD implementation because they were not designed for components written in Java.

Advancement in the area of a verifying compiler for industry-standard programming languages is also considerable but incomplete [6]. An influential verifying compiler for the Java programming language is the Extended Static Checker for Java (ESC/Java) [13]. Other successful prototype verifying compilers for Java, C#, and C are the KeY prover [14], VeriFast [19], Spec# [5], and VCC [10]. However, these verifying compilers use unique specification notations and thus cannot be used for the BDD component. For example, the KeY prover uses Java Modeling Language (JML) for specifications. To use these tools with the BDD component, the formal specifications would need to be reconstructed to match the required notation. Additionally, all of the library components used in the BDD component that were developed in the RESOLVE discipline would need to be replaced or modified to use the formal specifications of the new verifier. Another challenge is that many of these specification notations lack the clean semantics, full modularity, and comprehensibility of the RESOLVE framework, all of which ease verification.

3 Combining Value and Reference Semantics

3.1 RESOLVE and Value-Based Semantics

At the center of this project is the RESOLVE design philosophy and discipline for software components that allows for ease of use by clients, reusability of software, and the ability to formally verify both the software itself and the client code that uses it. This discipline describes the principles to design and compose high-quality component-based software systems. RESOLVE is also an integrated specification and programming *language* designed for building verified, component-based software. It is imperative and object-based and has a collection of components such as those found in the standard libraries for C++, C#, and Java. Programs written in RESOLVE can be verified with an automated prover. Figure 3.1 illustrates this automated verification on a List component for the `Reverse` procedure.

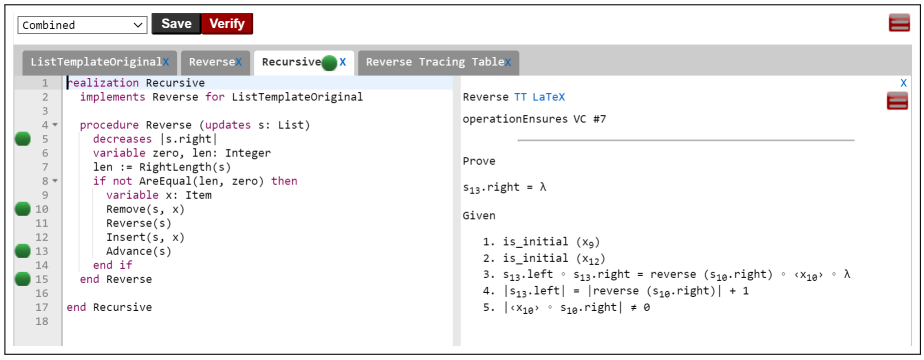

Fig. 1. A screenshot of the RESOLVE Web IDE

Parameter Modes An important construct in the RESOLVE discipline is the definition of *parameter modes* for arguments in method contracts. Parameter modes are used to define the modification frame of a method or loop. That is, they define whether a method or loop body can change the value of a formal parameter or variable. There are four parameter modes:

- **Clears:** The parameter is cleared to an initial value of its type
- **Updates:** The parameter can change value and the behavior of the method can depend on the parameter's (initial) value
- **Replaces:** The parameter can change value and the behavior of the method *does not* depend on the parameter's (initial) value
- **Restores:** (Default) The parameter's final value is the same as its initial value

Mathematical Model A mathematical model is an abstract definition of a component's state space. It is implementation-independent and defines a precise mathematical type that the client can use to reason about the component's behavior. A mathematical model is written in terms of base types or mathematical subtypes. Math base types are either basic types such as `integer`s or `boolean`s or composite types such as `tuple`s, `set`s, or `string`s (*i.e.*, sequences). Math subtypes are defined in terms of other math subtypes and base types. Math types are related to types in Java but are not equivalent. For example, the math type `integer` is infinite, but a Java `int` is bounded.

The math model for `BooleanStructure` is based on the mathematical type `BOOLEAN_STRUCTURE`. Clients use this math type to reason about the `BooleanStructure` component regardless of which implementation is used. The formal definition for this math type, shown in Listing 1.1, defines `BOOLEAN_STRUCTURE` as a pair containing a set of `ASSIGNMENT`s named *sat* and a string of integers named *vars*.

Listing 1.1. `BooleanStructure` Mathematical Model

```
1 /**
2  * @mathsubtypes
3  * ASSIGNMENT is finite set of integer
4  *
5  * BOOLEAN_STRUCTURE is (sat: finite set of ASSIGNMENT,
6  *                       vars: string of integer)
7  *
8  * @mathmodel type BooleanStructureKernel is modeled by
     BOOLEAN_STRUCTURE
9  */
```

Correspondence and Convention The `correspondence`, which is also referred to as the abstraction relation, defines how a particular implementation's specific representation, and similarly its `convention`, relates to the math model that applies to the general component interface that all implementations are based on. This relationship between the math model of the component and the mathematical representation of the concrete implementation allows a client to ignore the details of the implementation and reason about the component using only the mathematical model. Implementers are then able to reason about the implementing class using the mathematical representation that relates to the implementation details.

The `convention`, also referred to as the concrete invariant, defines constraints on a specific implementation. The `convention` for the `BooleanStructure` implementation verified in this work, shown in Listing 1.2, shows that the component has two constraints. The first is that the concrete field `$this.sat` does not contain any variables other than the ones in the field `$this.vars`. Note that the symbol "$" is used as a prefix to *this* to form `$this` when referring to the concrete state, in contrast to the abstract state which simply refers to `this`. The

second constraint is that the `$this.vars` field does not contain any duplicates. The `correspondence` for this component is trivial because it maps the concrete representation consisting of two fields, `$this.sat` and `$this.vars`, to the math model consisting of a tuple containing *sat* and *vars*.

Listing 1.2. `BooleanStructure` Convention and Correspondence

```
/**
 * @mathdefinitions
 *   NO_EXTRANEOUS_VARIABLES (
 *     s: set of ASSIGNMENT, t: string of integer
 *   ) : boolean satisfies
 *   for all a: ASSIGNMENT where ( a is in s )
 *     ( a is subset of entries(t) )
 *
 *   NO_DUPLICATES_IN_VARS (
 *     t: string of integer
 *   ) : boolean satisfies
 *   | t | = | entries(t) |
 *
 * @convention
 *     NO_EXTRANEOUS_VARIABLES($this.sat, $this.vars) and
 *     NO_DUPLICATES_IN_VARS($this.vars) and
 *
 * @correspondence this = ($this.sat, $this.vars)
 */
```

3.2 RESOLVE with Object-Oriented Languages

The RESOLVE discipline defines guidelines for developing high-quality and verifiable software and applies when reasoning about the behavior of programs. However, these discipline guidelines must be modified for use in practical programming languages such as Java or C++. For instance, the Java constructs of interfaces and classes are leveraged to accommodate the separation of abstract and concrete representations of a component necessary for the RESOLVE discipline [36]. Java components that use this discipline also commonly have more than one implementing class in the component, each with different time and space performance profiles but otherwise interchangeable from a client's perspective. In this way, each implementing class can have different concrete implementations, but the client reasons about them in the exact same way using the abstract representation, or math model, of the component. Another change to the discipline is necessary because of the risks introduced by the presence of inheritance in the Java programming language, since RESOLVE does not allow this capability. This change is the separation of component methods into *kernel* and *secondary* methods. Kernel methods are the minimal set of operations that allow the client to give a variable of the type any allowable value (controllability) and determine the value of a variable of the type (observability). Kernel methods

must be re-implemented for every implementing class in the component. Conversely, secondary layered methods are "layered" on top of kernel methods in an abstract class and are implemented as a client of the component, so they only need to be implemented once but apply to all implementing classes. An illustration of the RESOLVE component design pattern that separates the kernel and secondary methods of the BDD component is shown in Figure 2. This illustration also demonstrates the use of Java interfaces and classes to separate abstract and concrete elements of the component. The methods of the `BooleanStructure` component are also shown.

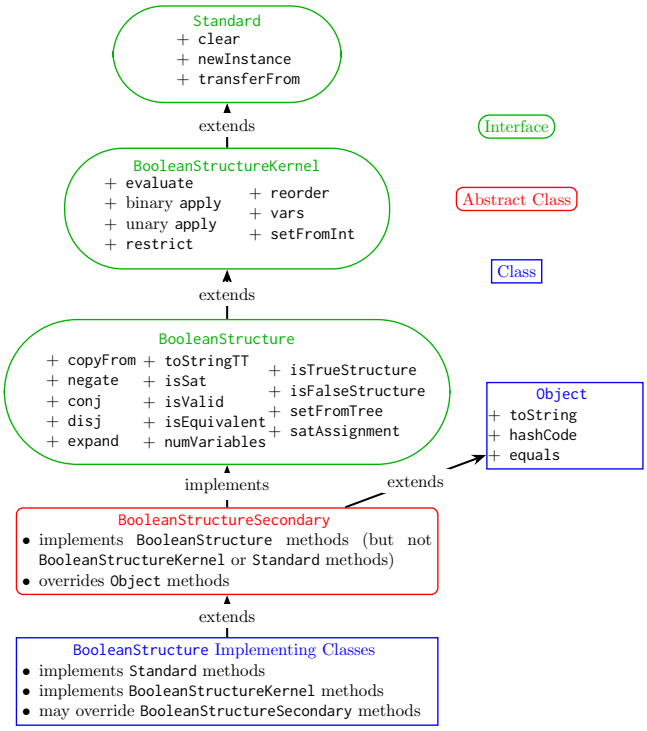

Fig. 2. `BooleanStructure` Component Diagram

The RESOLVE programming language has additional restrictions that are not present in Java and must also be accommodated. For example, RESOLVE uses *call-by-swapping* for parameter passing, unlike Java which allows references. RESOLVE also lacks an assignment operator, which prohibits aliasing, so formal verification in a Java component must verify that any aliases generated by the use of the assignment operator do not disrupt the soundness of the verification. The `BooleanStructure` component and all of the libraries used in it that follow the RESOLVE discipline attempt to avoid the pitfalls of the assignment opera-

tor by implementing the methods `transferFrom` and `copyFrom`. These methods are used in place of the assignment operator to transfer and copy objects while avoiding the complexities of aliasing. The BDD implementation of the `transferFrom` method assigns the concrete private fields of the BDD to those of the source BDD. The `copyFrom` method builds a copy of the source BDD by making copies of primitive types to avoid aliasing.

4 Formal Verification of the BDD Component

The full formal verification of a component involves the generation of loop invariants, reasoning tables, and proofs based on both the abstract and concrete components of the software. The goal of each proof in this work is to verify the correctness of a method in the `BooleanStructure` component. This verification requires proving that each method meets the requirements of its postcondition and eventually terminates. Details of the proof of correctness for this component are in [32].

4.1 Loop Invariants and Iteration

The construction of loop invariants allows loops to be traced in a verification proof without knowledge of how many times a loop will iterate during code execution. A special case of loop invariants is when they involve iterators, which require extra consideration in the verification process. An example of the for-each syntax in Java with a loop invariant can be seen in Listing 1.3. The loop in this example builds a duplicate of the sequence `this.vars` called `newOrder`. Correspondingly, the invariant for this loop maintains that the variable `newOrder` is always equal to the items in the collection `this.vars` that have already been iterated over. For verification purposes, the elements of the iterator that have been *seen* and *unseen* at a particular point in the loop can be accessed using the "∼" operator, such as in ∼`this.vars.seen`. The parameter mode for the variable ∼`this.vars` is *updates* because the seen and unseen elements are changing each iteration.

Listing 1.3. Example for-each Loop With Iterator

```
1 /**
2  * @updates newOrder, ~this.vars
3  * @maintains newOrder = ~this.vars.seen
4  * @decreases |~this.vars.unseen|
5  */
6 for (int elt : this.vars) {
7     newOrder.add(newOrder.length(), elt);
8 }
```

However, this for-each loop syntax requires additional effort in the construction of proofs. An example of the challenge associated with loops in the for-each format is that the loop body in Listing 1.3 cannot be used to prove that the

value of |~this.vars.unseen| decreases from one loop iteration to the next because the value of |~this.vars.unseen| is never explicitly updated through a call to an iterator's next method. Similarly, ~this.vars.seen is never explicitly updated within the loop. However, the behavior of a for-each loop dictates that the first operation of each loop iteration is to update this variable. Now, in the first line of the reasoning table for this loop, the value of ~this.vars.seen must simultaneously be its value before and after the update from an implicit call to next, which is undesirable.

To solve this problem, a strategy that maintains a close similarity to the source code but still allows for proofs of the loop invariant and progress metric is used, which is shown in Listing 1.4. This strategy is to add a comment containing a call to the iterator's next method at the beginning of the loop. This comment solves the problem of the .seen variable having two simultaneous values because in the reasoning table, .seen is updated after the commented call to next in line 7 of the listing. Thus, before line 7, the loop invariant is guaranteed to still hold, but after line 7 this is no longer a guarantee and the value of the seen and unseen elements of the collection have been updated.

Listing 1.4. Example for-each Loop With Iterator: Proof Equivalent

```
1 /**
2  * @updates newOrder, ~this.vars
3  * @maintains newOrder = ~this.vars.seen
4  * @decreases |~this.vars.unseen|
5  */
6 for (int elt : this.vars) {
7     // elt = this.vars.next();
8     newOrder.add(newOrder.length(), elt);
9 }
```

Due to the additional complexity of iterators, it is generally more desirable to use libraries of verified components with built-in functionality to perform tasks such as the one shown in Listing 1.3, which is to copy an object. These library components help avoid the dangers involved with iteration, such as the creation of aliases during iteration, but in this case the use of an iterator was unavoidable. Iteration over containers with immutable types, as in this example, does not threaten the validity of the proof, however.

4.2 Reasoning Tables and Proofs

A reasoning table is a method of organizing the facts and obligations, otherwise known as verification conditions, generated from the implementation body of a method. The facts and verification conditions are generated directly from specifications, the implementation, and the mathematical model, and the facts are used to confirm that the required verification conditions are met. The techniques employed in the production of the reasoning tables in this work use "natural reasoning" formulated by Heym [16] to aid in comprehensibility and usability. This reasoning technique is based on generating a sequence of facts that can

be combined to form new facts to prove verification conditions. An example of a reasoning table for the `BooleanStructure` method `apply`, which applies the unary operator *not*, can be seen in Table 1. Note that the initial facts in the reasoning table are the `convention` and `correspondence` from Listing 1.2. Other facts and verification conditions in the table are generated directly from pre- and postconditions of other component methods. The final verification conditions in the table are generated from the method postcondition, shown in Listing 1.5, and the restoration of the convention. The specification in Listing 1.5 refers to `#this`, where the "#" symbol refers to the state of the object before the method call and `this` refers to the final state.

Table 1. Unary `apply` Sample Reasoning Table

State	Path	Facts	Obligations
`public void apply(UnaryOperator op) {`			
		this = ($this.sat, $this.vars)	
0		NO_EXTRANEOUS_VARIABLES($this.sat$_0$, $this.vars$_0$)	
		NO_DUPLICATES_IN_VARS($this.vars$_0$)	
`if (op == UnaryOperator.NOT) {`			
1	op = NOT		
`Set<Set<Integer>> newSat = new Set2<Set<Integer>> ();`			
2	op = NOT	newSat$_2$ = {}	|$this.vars$_0$\| = \|entries($this.vars$_0$)|
`PowerStringElements allAssignments = new PowerStringElements(this.vars());`			
3	op = NOT	allAssignments = $this.vars$_0$ ~allAssignments.seen * ~allAssignments.seen = POWER_STRING(allAssignments) ~allAssignments.seen$_3$ =<>	entries(~allAssignments.seen$_3$) \ $this.sat$_0$ = newSat$_2$
`/**`			
`* @updates newSat, ~allAssignments`			
`* @maintains entries(~allAssignments.seen) \ $this.sat = newSat`			
`* @decreases | ~allAssignments.unseen|`			
`*/`			
`for (Set<Integer> a : allAssignments) {`			
4	op = NOT | ~allA...unseen$_4$| > 0	entries(~allAssignments.seen$_4$) \ $this.sat$_0$ = newSat$_4$	| ~allAssignments.unseen$_4$| > 0
`// a = allAssignments.next()`			
5	op = NOT	~allAssignments.seen$_5$ = ~allAssignments.seen$_4$ * <a> <a> * ~allAssignments.unseen$_5$ = ~allAssignments.unseen$_4$	NO_EXTRANEOUS_VARIABLES($this.sat$_0$, $this.vars$_0$) NO_DUPLICATES_IN_VARS($this.vars$_0$)
`if (!(processAssignment(this.sat, this.vars, a))) {`			
6	op = NOT not(a intersection...)		
`newSat.add(a);`			
7	op = NOT not(a intersection...)	newSat$_7$ = newSat$_4$ union {a}	
`} // end if`			
8	op = NOT	a intersection entries($this.vars$_0$) is in $this.sat$_0$ implies newSat$_8$ = newSat$_4$ not(a intersection entries($this.vars$_0$) is in $this.sat$_0$) implies newSat$_8$ = newSat$_7$	entries(~allAssignments.seen$_5$) \ $this.sat$_0$ = newSat$_8$ | ~allAssignments.unseen$_5$| < | ~allAssignments.unseen$_4$|
`} // end for`			
9	op = NOT	| ~allAssignments.unseen$_9$| = 0 entries(~allAssignments.seen$_9$) $this.sat$_0$ = newSat$_9$	NO_EXTRANEOUS_VARIABLES($this.sat$_9$, $this.vars$_0$)
`this.sat.transferFrom(newSat);`			
10	op = NOT	$this.sat$_{10}$ = newSat$_9$ newSat$_{10}$ = {}	
`} // end if`			
11		op = NOT implies $this.sat$_{11}$ = $this.sat$_{10}$ op /= NOT implies $this.sat$_{11}$ = $this.sat$_0$ $this.vars$_{11}$ = $this.vars$_0$	this.vars$_{11}$ = this.vars$_0$ for all p: ASSIGNMENT where (p is subset of entries(this.vars$_{11}$)) (p is in this.sat$_{11}$ iff ((if op = NOT then not(p intersection entries(this.vars$_0$) is in this.sat$_0$)) and (if op = IDENTITY then (p intersection entries(this.vars$_0$) is in this.sat$_0$)))) NO_EXTRANEOUS_VARIABLES($this.sat$_{11}$, $this.vars$_{11}$) NO_DUPLICATES_IN_VARS($this.vars$_{11}$)
`} // end unary_apply`			

Proofs of the verification conditions in the reasoning tables were constructed in a custom format but follow the general structure of a direct proof, otherwise known as a proof by construction. Facts taken directly from a reasoning table are the proof assumptions. A sequence of small, carefully justified steps

using these facts and mathematical axioms were used to construct new factual statements or lemmas. These steps are small enough to be mechanically checkable. This sequence eventually results in the verification condition for the proof. Proofs of each verification condition for the `BooleanStructure` component were constructed to form a single proof of correctness for the entire component.

Listing 1.5. Unary `apply` Specification

```
/**
 * Apply the unary operator {@code op} to {@code this}
 * without changing the total order of the variables of
 * {@code this}.
 *
 * @param op
 *   the unary operation to be applied on this
 * @updates this
 * @ensures
 *   this.vars = #this.vars and
 *   for all p: ASSIGNMENT where ( p is subset of
     entries(this.vars) )
 *      ( p is in this.sat iff
 *        ( ( if op = NOT then not EVALUATION(#this, p) ) and
 *          ( if op = IDENTITY then EVALUATION(#this, p) ) ) )
 */
void apply(UnaryOperator op);
```

5 Limitation in the Component Design Pattern

The Java component design pattern discussed in Section 3 is used in a sizeable component library which includes the BDD component verified in this work. During the process of verifying this BDD component, a shortcoming in the testing capabilities of this design pattern was discovered. This limitation was discovered in a component library used to teach thousands of computer science students since 2012 and was previously never discovered. This lack of detection indicates it is not an easily discoverable or obvious flaw. Additionally, it is likely that Java component design patterns similar to the one used in the `BooleanStructure` component also suffer from this testing limitation.

This limitation is related to how the design pattern uses abstract classes and overridden methods. Note that a component following the design pattern used in the `BooleanStructure` component may contain any number of implementing classes. For example, the BDD implementation illustrated in Figure 3 contains `BooleanStructure1` and `BooleanStructure2`. Also note that these implementing classes extend a single shared abstract class, `BooleanStructureSecondary`, containing the layered method implementations.

The component design pattern is organized so that implementing classes are interchangeable from a client's perspective. Further, implementing classes may override any number of the layered methods that are in the component's abstract

class. This is a desirable quality because overriding these methods allows their performance to be improved by leveraging direct access to concrete representation fields.

However, an undesirable consequence of this design pattern occurs when *every* implementing class overrides a particular layered method. Figure 3 illustrates this issue by listing which secondary methods are implemented in each class of the `BooleanStructure` component. Notice how four layered methods in `BooleanStructureSecondary`, including `expand`, `isTrueStructure`, `isFalseStructure`, and `satAssignment`, are overridden by all implementing classes of `BooleanStructure`. Since abstract classes cannot be instantiated directly, only instantiations of implementing classes can be tested by the test suite. As a consequence, if all of the implementing classes override a particular layered implementation, then that implementation never has the opportunity to be tested. This flaw is not critical because this untested code is by definition never used by any implementing classes so clients have no access to it. However, it is still undesirable to have untested and unreachable code in a component. Further, a future modification to the component may result in a new implementing class that does not override a previously hidden layered method, thus exposing the untested code and potentially an error.

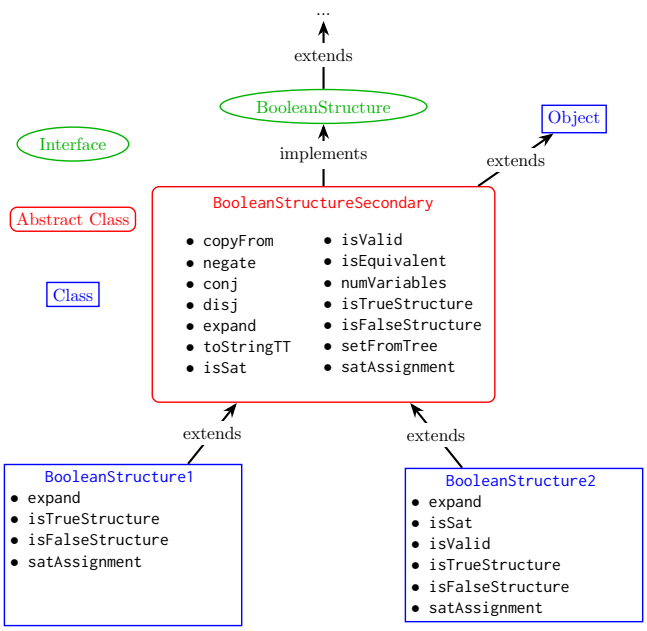

Fig. 3. `BooleanStructure` Secondary Method Implementations

To correct this limitation in the `BooleanStructure` component, the implementation was modified to include a new reference class that does not override

any secondary methods. This design pattern limitation is an interesting example of the challenges associated with verification using a language like Java that allows inheritance and does not have the strict limitations of the RESOLVE language.

6 Corrections to the Component

The unmodified BDD implementation contained 314 unit test cases in a test suite with 96.3% code coverage. Despite the high quality of this test suite, two implementation errors were discovered during the verification process. The detection of these errors despite a rigorous test suite demonstrates the relevance of the formal verification process. This also demonstrates how the verification process can be practically carried out on a Java component with RESOLVE specifications.

Additionally, many errors in the specifications were discovered in this process. Errors of this variety could lead to mistakes in client code due to a misrepresentation of `BooleanStructure` behavior. The errors of this type were also not revealed by the test suite.

`copyFrom` Runtime Exception An error in the behavior of the `copyFrom` method, which is shown in Listing 1.6, was discovered in the verification process. This error was likely not discovered previously because the method appears to be correct and passed many test cases with one-hundred percent code coverage. The `copyFrom` method is a secondary layered implementation, so it is not based on the underlying implementation of the component. In the original method implementation, a runtime exception occurs when there are no satisfying assignments in the method argument `BooleanStructure x` but the number of variables is nonzero. An example of a Boolean formula with this quality is $x_1 \wedge \neg x_1$ because it has no possible satisfying assignments but it is over a nonzero number of variables. In this scenario, the precondition for the `reorder` method that `VARIABLES(newExp) = entries(newOrder)` cannot be satisfied because the conditional block after `if (x.evaluate(t))` never executes when `x.sat` is empty. Thus, the variables in `newExp` remain in the initial empty state, causing a runtime exception in line 28 in the call to `reorder`.

Listing 1.6. `copyFrom` Original Implementation With Error

```
1  public void copyFrom(BooleanStructure x) {
2      // Generate a false structure with the same vars as x
3      BooleanStructure newExp = this.newInstance();
4      newExp.negate();
5      Sequence<Integer> order = x.vars();
6
7      // Take the disjunction of every assignment in x.sat
8      PowerStringElements allAssignments = new
           PowerStringElements(order);
9      for (Set<Integer> t : allAssignments) {
```

```
10          if (x.evaluate(t)) {
11              // Conjunct terms in t and negations in not(t)
12              BooleanStructure term = this.newInstance();
13              for (int v : order) {
14                  BooleanStructure vExp = this.newInstance();
15                  vExp.setFromInt(v);
16                  if (!t.contains(v)) vExp.negate();
17                  term.conj(vExp);
18              }
19              newExp.disj(term);
20          }
21      }
22
23      // Reorder variables in new structure to match x's order
24      Sequence<Integer> newOrder = new Sequence1L<Integer>();
25      for (int v : order) {
26          newOrder.add(newOrder.length(), v);
27      }
28      newExp.reorder(newOrder);
29
30      this.transferFrom(newExp);
31  }
```

`toStringTT` Violation of Postcondition The need for an additional constraint in a method precondition was revealed when the formal verification of the method `toStringTT` could not be completed. This method constructs a truth table representation of the BDD. The original implementation of this method performed a left bit shift operation based on the number of variables in the structure. However, since the `long` type in Java is limited to 64 bits, the method produced an erroneous result and violated the postcondition if the number of variables exceeded this limit. The added constraint in the precondition of the method requires that the number of variables is less than 64.

Inconsistent `isEquivalent` Math Definition The specification for the `isEquivalent` method, which compares the logical equivalence of two BDDs, had to be modified because the implementation and specification were inconsistent. The original specification compared the satisfying assignments and variables of the BDDs to check for equality. A modification to the specification was required because it did not consider logically equivalent BDDs with different variables to meet the specification, but the implementation did.

Missing `newOrder` Specifications The original `newOrder` private method, which constructs a variable ordering that is compatible with two input orderings, had a postcondition that was too weak to be useful. It required that the new ordering was compatible with the original two, but did not require anything about the

order entries. This is clearly too weak because an empty sequence would always satisfy this postcondition. To correct this, additional postcondition requirements were added to require that the new ordering variables are the union of the input variables. Further, the method required a strengthened precondition to prove a compatible ordering requirement of the original postcondition. In fact, a correct implementation of the specification without this precondition is impossible because a nontrivial compatible ordering of the result and the two inputs is impossible if the two inputs are not already compatible.

7 Conclusion

A contribution of this work is the identification of a limitation of the testing capabilities of the Java component design pattern used in the BDD implementation. The current structure of this design pattern allows for the possibility that all implementing classes of a component override the layered implementations of secondary methods, thus leaving these layered implementations unreachable and untested. This limitation demonstrates one of the challenges associated with combining an industry-standard language such as Java with formal verification.

A second outcome is the construction of a proof that formally verifies the correctness of a reference implementation of a Java-based BDD component. The resulting formally verified BDD component can now be used with a high level of confidence by clients. A third outcome resulting from the formal verification of the BDD component is the identification and correction of errors. Errors were discovered in both the specifications and implementation of the BDD component. The errors discovered in the implementation are particularly notable because they were not discovered by the comprehensive test suite. These errors were discovered only in the formal verification process, which indicates how critical formally verifying software is for error-resistant software development.

An expansion of this work is to develop an automated theorem prover to automate the verification process of a Java-based component with RESOLVE specifications. This verifier would be uniquely practical and useful because of the use of an industry-standard programming language with a specification notation that is particularly well-suited to client reasoning and modularity. This project lays some groundwork for an automated prover of this type because it provides a carefully constructed example of valid inputs and a corresponding ideal expected output. A verifier for a Java-based component with RESOLVE specifications would require the construction of a tool to automate the generation of verification conditions in a modular fashion. Existing RESOLVE verifiers [33, 17] could then be leveraged with only slight modifications to discharge a substantial proportion of the verification conditions in an automated way.

Acknowledgement. The authors would like to acknowledge Saad Asim for his development of the original BDD code base. Additionally, this work has benefited from extensive discussions with other the members of the Reusable Software Research Group.

References

1. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. SIGPLAN Not. **37**(11), 311–330 (Nov 2002). https://doi.org/10.1145/583854.582448
2. Asim, S.: An exercise in design: The binary decision diagram. SIGSOFT Softw. Eng. Notes **43**(3), 19 (Dec 2018). https://doi.org/10.1145/3229783.3229801, https://doi.org/10.1145/3229783.3229801
3. Asim, S.: The Binary Decision Diagram: Abstraction and Implementation. Master's thesis, The Ohio State University (2018)
4. Aziz, A., Balarin, F., Cheng, S., Hojati, R., Kam, T., Krishnan, S., Ranjan, R., Shiple, T., Singhal, V., Tasiran, S., Wang, H., Brayton, R., Sangiovanni-Vincentelli, A.: HSIS: a BDD-based environment for formal verification. In: 1994 31st Design Automation Conference. pp. 454–459. IEEE Computer Society, Los Alamitos, CA, USA (June 1994). https://doi.org/10.1145/196244.196467, https://doi.ieeecomputersociety.org/10.1145/196244.196467
5. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and Verification: The Spec# Experience. Commun. ACM **54**(6), 81–91 (Jun 2011). https://doi.org/10.1145/1953122.1953145
6. Beyer, D.: Advances in Automatic Software Verification: SV-COMP 2020. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 347–367. Springer International Publishing, Cham (2020)
7. Brookes, S.: A semantics for concurrent separation logic. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory. pp. 16–34. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
8. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic model checking: 10^{20} States and beyond. Information and Computation **98**(2), 142–170 (1992). https://doi.org/https://doi.org/10.1016/0890-5401(92)90017-A
9. Chaki, S., Gurfinkel, A.: BDD-Based Symbolic Model Checking, pp. 219–245. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_8
10. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Theorem Proving in Higher Order Logics. pp. 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
11. van Dijk, T., Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer **19** (11 2017). https://doi.org/10.1007/s10009-016-0433-2
12. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: Programming Languages and Systems. pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. SIGPLAN Not. **37**(5), 234–245 (May 2002). https://doi.org/10.1145/543552.512558
14. Hähnle, R., Menzel, W., Schmitt, P.H.: Integrierter deduktiver software-entwurf. Künstliche Intell. **12**(4), 40–41 (1998)
15. Herbert, L., Leino, K.R.M., Quaresma, J.: Using Dafny, an Automatic Program Verifier, pp. 156–181. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35746-6_6

16. Heym, W.: Computer Program Verification: Improvements for Human Reasoning. Ph.D. thesis, Ohio State University (1995)
17. Hoffman, D.: Techniques for the Specification and Verification of Enterprise Applications. Ph.D. thesis, Ohio State University (2016)
18. Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R.: The geneva convention on the treatment of object aliasing. SIGPLAN OOPS Mess. **3**(2), 11–16 (Apr 1992). https://doi.org/10.1145/130943.130947
19. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: NASA Formal Methods. pp. 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
20. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, USA (2018)
21. Kulczycki, G., Sitaraman, M., Ogden, W., Leavens, G.: Preserving clean semantics for calls with repeated arguments. Technical Report RSRG-04-01, Department of Computer Science, Clemson University (April 2003), `http://www.cs.clemson.edu/~resolve`
22. Kulczycki, G., Vasudeo, J.: Simplifying reasoning about objects with tako. In: Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems. p. 57–64. SAVCBS '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1181195.1181207
23. Kurai, R., Minato, S.i., Zeugmann, T.: N-Gram Analysis Based on Zero-Suppressed BDDs. In: New Frontiers in Artificial Intelligence. pp. 289–300. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
24. Lee, C.Y.: Representation of switching circuits by binary-decision programs. The Bell System Technical Journal **38**(4), 985–999 (1959). https://doi.org/10.1002/j.1538-7305.1959.tb01585.x
25. Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/` (2003)
26. Loekito, E., Bailey, J., Pei, J.: A binary decision diagram based approach for mining frequent subsequences. Knowledge and Information Systems **24**(2), 235–268 (2010)
27. Lovato, A., Macedonio, D., Spoto, F.: A thread-safe library for binary decision diagrams: Software Engineering and Formal Methods. Springer International Publishing, 1st edn. (2014)
28. Lv, G., Su, K., Xu, Y.: CacBDD: A BDD Package with Dynamic Cache Management. In: Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044. p. 229–234. CAV 2013, Springer-Verlag, Berlin, Heidelberg (2013)
29. Minsky, N.H.: Towards alias-free pointers. In: Cointe, P. (ed.) Proceedings ECOOP '96 - Object-Oriented Programming. pp. 189–209. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
30. Pearce, D.J., Groves, L.: Designing a verifying compiler: Lessons learned from developing Whiley. Science of Computer Programming **113**, 191–220 (2015). https://doi.org/https://doi.org/10.1016/j.scico.2015.09.006, formal Techniques for Safety-Critical Systems
31. Qian, F.: SableJBDD: a Java Binary Decision Diagram Package. `http://www.sable.mcgill.ca/~fqian/SableJBDD/` (2004)
32. Rumreich, L.: The Binary Decision Diagram: Formal Verification of a Reference Implementation. Master's thesis, The Ohio State University (2021)
33. Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H.M., Harton, H., Heym, W., Kirschenbaum, J., Krone, J.,

Smith, H., Weide, B.W.: Building a Push-Button RESOLVE Verifier: Progress and Challenges. Formal Aspects of Computing **23**(5), 607–626 (Sep 2011). https://doi.org/10.1007/s00165-010-0154-3
34. Sitaraman, M., Weide, B.: Component-based software using RESOLVE. ACM SIGSOFT Software Engineering Notes **19**(4), 21–22 (1994). https://doi.org/10.1145/190679.199221
35. Sivilotti, P., Asim, S., Rumreich, L.: BDD: A Binary Decision Diagram Software Component. `https://github.com/osu-rsrg/BDD` (2021)
36. Sivilotti, P.A., Lang, M.: Interfaces First (and Foremost) with Java. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education. p. 515–519. SIGCSE '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1734263.1734436
37. Somenzi, F.: CUDD: CU decision diagram package release 3.0.0. `http://vlsi.colorado.edu/fabio/CUDD/` (2015)
38. Vahidi, A.: JDD: a pure Java BDD and Z-BDD library. `https://bitbucket.org/vahidi/jdd/src/master/` (2019)
39. Xing, L.: An Efficient Binary-Decision-Diagram-Based Approach for Network Reliability and Sensitivity Analysis. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans **38**(1), 105–115 (2008). https://doi.org/10.1109/TSMCA.2007.909493
40. Zweben, S.H., Edwards, S.H., Weide, B.W., Hollingsworth, J.E.: The effects of layering and encapsulation on software development cost and quality. IEEE Transactions on Software Engineering **21**(3), 200–208 (1995). https://doi.org/10.1109/32.372147