AIOOL 2005 Preliminary Version

Decision Procedures for Set-Valued Fields

Viktor Kuncak [1] Martin Rinard [2]

MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, Massachusetts, USA

Abstract

A central feature of current object-oriented languages is the ability to dynamically instantiate user-defined container data structures such as lists, trees, and hash tables. Implementations of these data structures typically use references to dynamically allocated objects, which complicates reasoning about the resulting program. Reasoning is simplified if data structure operations are specified in terms of abstract sets of objects associated with each data structure. For example, an insertion into a data structure in this approach becomes simply an insertion into a dynamically changing set-valued field of an object, as opposed to a manipulation of a dynamically linked structure attached to the object.

In this paper we explore reasoning techniques for programs that manipulate data structures specified using set-valued abstract fields associated with container objects. We compare the expressive power and the complexity of specification languages based on 1) decidable prefix vocabulary classes of first-order logic, 2) two-variable logic with counting, and 3) Nelson-Oppen combinations of multisorted theories. Such specification logics can be used for verification of object-oriented programs with supplied invariants. Moreover, by selecting an appropriate subset of properties expressible in such logic, the decision procedures for these logics enable automated computation of lattice operations in an abstract interpretation domain, as well as automated computation of abstract program semantics.

Key words: Program verification, Data structures, Objects, Decision procedures, Two-variable logic with counting, Classical decision problem, Nelson-Oppen technique, Two-level syllogistic

1 Introduction

Analysis and verification of modern object-oriented programming languages poses unique challenges [29, 18, 24, 11]. In this paper we study a central feature of current object-oriented languages: the ability to introduce user-defined abstract data types, and create an unbounded number of instances of these data types during program execution. Particular difficulties arise when each data type instance is itself implemented using multiple dynamically allocated objects that form a linked data structure. Our approach for analyzing such

[1] Email: vkuncak@csail.mit.edu

[2] Email: rinard@csail.mit.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

programs is to use abstract set-valued fields as specification variables that describe operations of an abstract data type. We separate the analysis of the program into 1) verifying the correctness of the implementation of the abstract data type with respect to the set specification, and 2) verifying the correctness of the rest of the program where the linked data structure is replaced by abstract set-valued fields.

Hob project. One of the main design principles behind the Hob project [17, 32] is that reasoning about programs with complex data structures becomes simpler if data structure operations are specified in terms of abstract sets of objects associated with each data structure. For example, an insertion into a data structure in this approach becomes simply an insertion into a dynamically changing sets of objects, as opposed to a manipulation of a dynamically linked data structure. Hob splits the verification of programs with such data structures into two tasks: 1) using shape analysis (or some other analysis or verification technique, including techniques as powerful and heavyweight as interactive theorem proving) to verify that the data structure implementation conforms to the specification given in terms of the abstract set variables, and 2) using only the abstract set variables in the rest of the program to reason about the behavior of the data structure. So far, we have used Hob to verify implementations of global data structures, which are instantiated at compile time into a finite number of instances (that may store and manipulate a statically unbounded number of objects). The focus on global data structures allowed us to use a static module mechanism to encapsulate object fields and prevent representation exposure, as well as to use the decidable theory of Boolean algebras [13] to reason about the finite number of abstract sets that specify data structures. One goal of this research is to extend this approach for dynamically instantiated data structures as well.

Dynamic instantiation of linked data structures. Dynamic instantiation of abstract data types is one of the central features of current object-oriented programming languages. Dynamic instantiation is typically achieved by associating each abstract data type instance with an object, using a field to attach the underlying linked data structure to the object. This research explores one way to extend Hob to verify programs that use linked data structures that can be dynamically instantiated. In this approach, we specify a linked data structure attached to an object using a finite number of set-valued object fields. The result of abstracting the content using this technique is a program that manipulates objects connected using relations. A relation in the resulting program can be either a function (whose value for a given object is the object referenced by an object-valued field), or a general relation (whose value for a given object is the set of objects stored in the data structure associated with the object).

The generalization to dynamic instantiation of data structures in Hob requires extensions to both phases of verification: 1) verification that the linked data structure conforms to the set interface given by values of object fields and 2) verification of the resulting program that uses objects with set-valued fields. To address the first phase, we are extending the existing technique in Hob with the techniques for specifying representations of individual objects [22, 6, 5, 2, 3]; these extensions are necessary to ensure that the analysis of one instance remains valid in the presence of other instances in the heap.

The topic of this paper is the second problem: verification of programs that manipulate objects with set-valued fields. Like [26], we are concerned with verification of clients of abstract data types, but we focus on specifications expressed in terms of set-valued fields and derive a complete decision procedure for the constraints in our class. Our approach uses assume/guarantee reasoning with user-supplied annotations to completely separate the analysis of the implementation of the class from the analysis of the context; other approaches attempt to automatically infer both the approximation of the context and the approximation of class implementation [18], potentially using a global fixpoint analysis.

Decision procedures for set-valued fields. To study the automation of reasoning about programs with set-valued fields, we explore decision procedures for constraints on such fields. Our constraints can express relationships between sets associated with the same object, the aliasing between object references, as well as the relationships between sets associated with different objects. By annotating programs with such constraints and using a verification-condition generator [32], developers can verify a range of invariants of object-oriented programs. By selecting an appropriate subset of properties expressible using such constraints, a decision procedure for these constraints enables an analysis to automatically derive the lattice operations in the abstract interpretation domain, and to compute abstract program semantics (transfer functions).

Contributions and overview. To motivate the constraints studied in this paper, we present an example in Section 2. We present our formal setup in Section 3. As the main result of this paper, we explore reasoning techniques for programs that use set-valued abstract fields. We compare the expressive power and the complexity of specification languages based on decidable prefix classes of first-order logic (Section 5), two-variable logic with counting (Section 6), and Nelson-Oppen combinations of multisorted theories (Section 7). We observe that both the decidable prefix class $[\exists^*\forall^*]_=$ and Nelson-Oppen combination yield optimal NP algorithms for deciding an interesting class of constraints. On the other hand, the use of two-variable logic with counting allows more expressive constraints (such as the constraint that a field is never null), but requires an NEXPTIME decision procedure in general. We present conclusions in Section 8 and discuss related work throughout the paper.

2 Example

Figure 1 presents an example program fragment containing a precondition (expressed using an assume statement), a loop invariant (expressed using [. . .] brackets just before the condition of the while loop), and a postcondition (expressed using an assert statement). The program fragment empties the set $x.\mathsf{c}$ and copies its content into the set $y.\mathsf{c}$ (one could imagine some processing of primitive fields of e being performed in each loop iteration, but this is of

$\mathsf{assume}\ x \neq \mathsf{null} \wedge x \in \mathsf{alloc};$
$\mathsf{oldxc} := x.\mathsf{c};$
$\mathsf{new}\ y;$
$\mathsf{while}\ \ [x \neq \mathsf{null} \wedge y \neq \mathsf{null} \wedge x \neq y \wedge x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc}]$
$\quad (x.\mathsf{c} \neq \emptyset)$
$\{$
$\quad e := \mathsf{removeFirst}(x);$
$\quad //\ \mathsf{process}(e);$
$\quad \mathsf{insert}(y, e);$
$\}$
$\mathsf{assert}\ y.\mathsf{c} = \mathsf{oldxc};$

Fig. 1. An example program fragment that manipulates set-valued fields. Here $z.\mathsf{c}$ denotes the value of the set associated with object denoted by z

$e := \mathsf{removeFirst}(x)$:
$\quad \mathsf{assert}\ x.\mathsf{c} \neq \emptyset;$
$\quad \mathsf{havoc}\ e;$
$\quad \mathsf{assume}\ e \in x.\mathsf{c};$
$\quad x.\mathsf{c} := x.\mathsf{c} \setminus \{e\}$

$\mathsf{insert}(y, e)$:
$\quad y.\mathsf{c} := y.\mathsf{c} \cup \{e\}$

Fig. 2. Specifications of procedure calls from Figure 1

$$\begin{array}{l} x \neq y \ \wedge \ x \neq \mathsf{null} \ \wedge \ y \neq \mathsf{null} \ \wedge \ x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc} \ \Rightarrow \\ \qquad e \in x.\mathsf{c} \Rightarrow x \neq y \ \wedge \ (x.\mathsf{c} \setminus \{e\}) \cup (y.\mathsf{c} \cup \{e\}) = \mathsf{oldxc} \end{array}$$

Fig. 3. Verification condition for loop preservation of example in Figure 1

$$\begin{array}{rcl} O & ::= & V_O \mid \mathsf{null} \mid O.f_O \\ S & ::= & V_S \mid S_1 \cup S_2 \mid S_1 \cap S_2 \mid S_1 \setminus S_2 \mid \{O_1, \ldots, O_n\} \mid O.f_S \\ f & ::= & V_f \mid f_O[O_1 \mapsto O_2] \mid f_S[O \mapsto S] \\ A & ::= & O_1 = O_2 \mid O \in S \mid S_1 = S_2 \mid \mathsf{card}(S) \leq k \mid f_1 = f_2 \\ F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \end{array}$$

Fig. 4. Syntax of expressions and formulas

no relevance to our example). The property that we wish to verify is that the content of the set $y.\mathsf{c}$ at the end of the program fragment is equal to the original content of the set $x.\mathsf{c}$, which is stored in the auxiliary set-valued local variable oldxc. The property is true, because procedure call removeFirst removes an element from $x.\mathsf{c}$ and returns it in e, and then insert inserts the same element into $y.\mathsf{c}$. Figure 2 shows guarded-command specifications of procedure calls that we use to reason about the effects of procedures; our system verifies separately that procedures conform to their specifications.

Given the precondition, loop invariant and the postcondition for the program fragment in Figure 1, we can generate verification conditions that imply that the program postconditions will hold. Figure 3 shows one of the verification conditions for program in Figure 1. Note that the resulting constraints require not only reasoning about the content of individual sets (as in the semantics of insert), but also reasoning about aliasing of references to objects (as in the conjunct $x \neq y$) and reasoning about the relations between sets associated with distinct objects (as in the conjunct $x.\mathsf{c} \cup y.\mathsf{c} = \mathsf{oldxc}$). In Section 3 we define a class of such constraints on objects with set-valued fields; in the rest of this paper we study the validity and the satisfiability problem for constraints in this class.

3 Specification Language

Figure 4 shows the syntax of our specification language for expressing constraints on objects with set-valued fields. Our specification language is typed (multisorted); we are only concerned with well-typed formulas. The nonterminal O denotes objects, which can be potentially null, S denotes sets of non-null objects, and f denotes fields. Fields can map objects to objects (then they are denoted f_O) or they can map objects to sets (then they are denoted f_S). We use formulas (the non-terminal F) as part of assume and assert statements, the conditions of while loops, and if statements. We use the object-valued and set-valued terms of this language (the non-terminals O and S in Figure 4) on the right-hand side of the assignment statements. The meaning of constructs in this language is straightforward. Notation $x.f$ denotes a dereference of a field f of objects x, which can be thought of as a function application that signals an error if the object x is null. Notation $f_O[o_1 \mapsto o_2]$ denotes an update of an object-valued field f_O so that $o_1.f_O = o_2$ and the value of the same field for all other objects is the same; such update operation corresponds to an array update if we view the field f as an array of objects indexed by objects. Set operations in our language have standard meaning. In the expression $\mathsf{card}(S) \leq k$, notation $\mathsf{card}(S)$ denotes the number of elements (cardinality) of the set S, and k denotes a non-negative integer constant. For complexity considerations, note that we represent integer constants in unary notation, so a constant k has the length k as opposed to $\log k$.

This paper considers decision procedures for the validity of formulas whose syntax is given by non-terminal F in Figure 4. The validity of such formulas can be used to show the validity of verification conditions in a programming language. Deciding verification conditions allows verification of object-oriented programs annotated with loop invariants. A decision procedure can also be used to synthesize loop invariants by identifying a finite set of formulas that forms an abstract domain. If $\mathcal{S}$ is a finite set of predicates, let $D(\mathcal{S})$ denote the set of all disjunctions of formulas in $\mathcal{S}$ and let $B(\mathcal{S})$ denote the set of all Boolean combinations of formulas in $\mathcal{S}$. If $\mathcal{S}$ is a finite set of closed formulas in language of Figure 4, then the sets $D(\mathcal{S}) \cup \{\mathsf{true}\}$ and $B(\mathcal{S})$ are also formulas in language of Figure 4 and can be used as domains. Transfer functions for such domains can be computed using predicate abstraction [1]. Let $Q_{v_1,\ldots,v_n}(\mathcal{S}) = \{\forall v_1, \ldots, v_n.\ F \mid F \in \mathcal{S}\}$ and let $\mathcal{S}$ be a set of quantifier-free formulas with unary and binary function symbols. Then $D(Q_{v_1,\ldots,v_n}(B(\mathcal{S})))$ is a domain of formulas that can be used for symbolic shape analysis [30]. Satisfiability of formulas in class $D(Q_{v_1,\ldots,v_n}(B(\mathcal{S})))$ as well as $B(Q_{v_1,\ldots,v_n}(B(\mathcal{S})))$ can decided using the $[\exists^*\forall^*]_=$ class of formulas used in Section 5.

4 Preliminary Observations

We first make several observations on deciding the validity of formulas in the language of Figure 4. Note that the language is closed under all boolean operations, so we only consider the satisfiability problem. Our constraints are quantifier-free, so we can view the satisfiability algorithm as a nondeterministic procedure that selects a satisfying assignment to atoms of the quantifier-free formula, and checks that the satisfying assignment corresponds to a satisfiable conjunction of literals [8]. In this way we reduce satisfiability of constraints to satisfiability of conjunctions of literals. Note finally that we can transform every conjunction of literals into an equisatisfiable conjunction that contains no nested terms. We transform a formula into such unnested form by introducing fresh variables; these fresh variables become existentially quantified, because we are looking at satisfiability. In the resulting unnested form, each atomic formula is of one of the following syntactic forms: $V_O^1 = V_O^2.f_O$, $V_S = V_S^1 \cup V_S^2$, $V_S = V_S^1 \cap V_S^2$, $V_S = V_S^1 \setminus V_S^2$, $V_S = \{V_O^1, \ldots, V_O^n\}$, $V_S = V_O.f_S$, $V_f^1 = V_f^2[V_O^1 \mapsto V_O^2]$, $V_f^1 = V_f^2[V_O \mapsto V_S]$, $V_O^1 = V_O^2$, $V_O \in V_S$, $V_S^1 = V_S^2$, $\mathsf{card}(V_S) \leq k$, $V_f^1 = V_f^2$. In the sequel we outline decidability of conjunctions of such unnested formulas and their negations. We consider three different methods. We pay most attention to the first method (Section 5).

5 A Classical Prefix-Vocabulary Class

In this section we outline our first technique for checking satisfiability of conjunctions of unnested literals. This technique is based on the class of universal formulas in first-order logic with a relational signature, without function symbols of non-zero arity. We translate conjunctions of literals into equisatisfiable formulas in this class while introducing a bounded number of universal quantifiers (namely, at most three).

The class $[\exists^*\forall^*]_=$. Define the class $[\exists^*\forall^q]_=$ as the set of all formulas of the form $\exists x_1, \ldots, x_p. \forall y_1, \ldots, y_q.F$ where $p \geq 0$ and F is quantifier-free formula of first-order logic with equality without function symbols. Let $[\exists^*\forall^*]_=$ be the set of formulas $\bigcup_{q \geq 0} [\exists^*\forall^q]_=$. We then have the following [4, Page 258].

Fact 5.1 *For any fixed q, satisfiability for $[\exists^*\forall^q]_=$ is in NP.*

Fact 5.2 *The satisfiability for $[\exists^*\forall^*]_=$ is in NEXPTIME.*

The idea of the translation. The translation of the language in Figure 4 into $[\exists^*\forall^*]_=$ class can be summarized as follows: 1) use unary relations to represent sets, 2) use binary relations to represent object-valued and set-valued fields, and 3) use universal quantifiers to represent set operations. To make this approach work, we need to properly represent null references, eliminate array updates by case analysis, and carefully translate cardinality constraints to avoid introducing an unbounded number of $\forall$ quantifiers. To ensure that object-valued fields are not assigned multiple values simultaneously, for each object-valued field f_O we introduce a conjunct

$$\forall x, y, z.\ f_O(x, y) \wedge f_O(x, z) \Rightarrow y = z \tag{1}$$

We would like to consider only models where fields are total functions that potentially have the value null. The $[\exists^*\forall^*]_=$ fragment cannot ensure this invariant, but we can transform formula into a form such that the existence of a model that does not satisfy this invariant implies the existence of a model that satisfies the invariant (see [16] for details). The main rules for the translation of positive literals are in Figure 5. To translate a negative literal, negate the translation of the underlying atomic formula as in Figure 5, replacing universal quantifiers with existential quantifiers, and then drop existential quantifiers while making sure that the newly introduced variables are fresh.

F	$[\![F]\!]$
$V_O^2 = V_O^1.f_O$	$f_O(V_O^1, V_O^2)$
$V_S = V_S^1 \cup V_S^2$	$\forall^+ x.\ V_S(x) \iff V_S^1(x) \lor V_S^2(x)$
$V_S = V_S^1 \cap V_S^2$	$\forall^+ x.\ V_S(x) \iff V_S^1(x) \land V_S^2(x)$
$V_S = V_S^1 \setminus V_S^2$	$\forall^+ x.\ V_S(x) \iff V_S^1(x) \land \neg V_S^2(x)$
$V_S = \{V_O^1, \ldots, V_O^n\}$	$\forall^+ x.\ V_S(x) \iff x = V_O^1 \lor \ldots \lor x = V_O^n$
$V_S = V_O.f_S$	$\forall^+ x.\ V_S(x) \iff f_S(V_O, x)$
$V_O^1 = V_O^2$	$V_O^1 = V_O^2$
$V_O \in V_S$	$V_S(V_O)$
$V_S^1 = V_S^2$	$\forall^+ x.\ V_S^1(x) \iff V_S^2(x)$
$V_f^1 = V_f^2$	$\forall^+ x, y.\ V_f^1(x,y) \iff V_f^2(x,y)$
$V_f^1 = V_f^2[V_O^1 \mapsto V_O^2]$	$\forall^+ x, y.\ V_f^1(x,y) \iff ((x = V_O^1 \land y = V_O^2) \lor (x \neq V_O^1 \land V_f^2(x,y)))$
$V_f^1 = V_f^2[V_O \mapsto V_S]$	$\forall^+ x, y.\ V_f^1(x,y) \iff ((x = V_O \land V_S(y)) \lor (x \neq V_O \land V_S(x,y)))$

Fig. 5. Rules for transforming positive literals into $[\exists^*\forall^*]_=$ fragment

We translate positive cardinality constraint $\mathsf{card}(V_S) \leq k$ by introducing k fresh constants $a_1, \ldots, a_k$, replacing the constraint with $V_S = \{a_1, \ldots, a_k\}$, and then translating the result as in Figure 5. We translate the negative cardinality constraint $\neg(\mathsf{card}(V_S) \leq k)$, which is equivalent to $\mathsf{card}(V_S) \geq k+1$, by introducing fresh constants $a_1, \ldots, a_k, a_{k+1}$, and replacing the constraint with $\bigwedge_{1 \leq i \leq k+1} V_S(a_i) \ \land \ \bigwedge_{1 \leq i < j \leq k+1} a_i \neq a_j$.

Complexity. We next show that satisfiability of formulas in Figure 4 is NP complete. We have carefully constructed our translation so that it introduces a bounded number of quantifiers. Indeed, each conjunct introduces at most three universal quantifiers (three quantifiers are needed for (1)). By moving these quantifiers to prenex position we can write the formula in form $[\exists^*\forall^3]_=$. Because the size of the generated formula is polynomial in the size of the original formula and the time to generate it is polynomial, by Fact 5.1 we conclude that checking the satisfiability of one assignment to unnested atomic formulas is in NP. Unnested form is polynomial in the size of conjunction of literals that specifies an assignment to atomic formulas of a formula F in Figure 4, and picking an assignment to atomic formulas can be done in NP. By composing these two non-deterministic choices, we obtain an NP decision procedure for satisfiability of expressions. NP-hardness follows because our language subsumes propositional logic. We conclude that the satisfiability of formulas in Figure 4 is NP-complete.

Remarks on related work. Fragments of first-order logics based on quantifier prefixes are systematized in [4, 9] where the $[\exists^*\forall^*]_=$ class is described as Bernays-Schönfinkel-Ramsey class. Finite model finding tools such as Alloy [12] can therefore be used to check satisfiability of such formulas. Resolution techniques [27] are also complete for this class because the term model is finite.

6 Two-Variable Logics

In this section we show that two-variable logic with counting, denoted C^2, can be used to decide constraints in Figure 4, as well as some useful extensions of these constraints. We consider the satisfiability problem and use a language containing any number of constants, unary relation symbols, and binary relation symbols.

Two-variable logics. The logic C^2 is a first-order logic 1) extended with counting quantifiers $\exists^{\geq k} x.F(x)$, saying that there are at least k elements x satisfying formula $F(x)$ for some constant k, and 2) restricted to allow only two variable names x, y in formulas. Note that the variables x and y may be reused via quantifier nesting, and that formulas of the form $\exists^{=k} x.\, F(x)$ and $\exists^{\leq k} x.\, F(x)$ are expressible as boolean combination of formulas of the form $\exists^{\geq k} x.\, F(x)$. The logic C^2 was shown decidable in [10] and the complexity for the C^2_1 fragment of C^2 was established in [23, 25]. The two-variable logic without counting L^2 has finite model property [19]; this is not the case for two-variable logic with counting [10]. The usefulness of two-variable logic with counting for reasoning about relations between objects in imperative programs was identified in [14, 15].

Encoding into two-variable logic with counting. Consider the problem of encoding the constraints in Figure 4 into two-variable logic with counting. It turns out that most of the ideas of the encoding in Section 5 apply to encoding using two-variable logic as well, because we only use at most two universal quantifiers in Figure 5, and the existentially quantified variables simply become constants in the language. To avoid using three variables to express the fact that some relations are functions, we replace (1) with $\forall x.\exists^{\leq 1} y.f(x, y)$. We can express the cardinality constraints directly by replacing $\mathsf{card}(S) \leq k$ with $\exists^{\leq k} x.S(x)$.

7 Nelson-Oppen Combination

We next note that satisfiability of formulas in Figure 4 can be decided using a multi-sorted version of the Nelson-Oppen decision procedure that combines three individual decision procedures: 1) theory for reasoning about sets expressed as a component Nelson-Oppen procedure, similarly to [31], but with addition of finite cardinalities (represented using ideas from Section 5); 2) uninterpreted function symbols [21] in multisorted language with function symbols whose result sort can be a set sort; and 3) extensional theory of arrays [20,28]. Because an equivalence class on shared variables in Nelson-Oppen decision procedure can be guessed using an NP algorithm, and each individual decision procedure is in NP, we conclude that a Nelson-Oppen combination decision procedure for our language is also in NP. While [31] already observes that reasoning about elements can be combined with reasoning about sets of elements using Nelson-Oppen procedure, we here observe the usefulness of combining the resulting procedure with uninterpreted function symbols and arrays, obtaining a decision procedure for reasoning about set-valued fields of objects.

8 Discussion and Concluding Remarks

We have outlined three techniques for solving constraints on set-valued fields: reduction to $[\exists^*\forall^*]_=$ class of first-order logic, reduction to two-variable logic with counting, and the use of Nelson-Oppen combination for multi-sorted theories. If the goal is only to decide the language in Figure 4, then both $[\exists^*\forall^*]_=$ class and Nelson-Oppen combination yield optimal decision procedure. Two-variable logic has more complex decision procedure, but also has the additional expressive power that enables expressing the constructs of the form $\forall x.\exists y.f(x,y)$. Such constructs allow us to state non-null properties of objects, which are important for reasoning about initialization of objects in object-oriented programming languages [7]. Moreover, counting quantifiers can naturally express high-level application constraints identified in the database community and object-oriented modelling community as referential integrity and cardinality constraints, as well as role constraints [15]. Note that $[\exists^*\forall^*]_=$ fragment is also more expressive than the language in Figure 4, allowing the statement of properties such as $\forall x, y.A(x) \Rightarrow (f(x,y) \Leftrightarrow g(y,x))$ while retaining the bounded number of quantifiers and therefore an NP decision procedure. The approach based on decomposing the language of Figure 4 into smaller Nelson-Oppen theories as in Section 7 has the advantage of using previously understood and efficient decision procedures that may be useful in other contexts. Moreover, no special encodings are necessary because the use of sorts naturally decomposes constraints into the constraints of individual decidable theories. Our observations imply that each of the individual theories in Nelson-Oppen combination can be showed decidable using the result of Section 5. Finally, we can use the idea of Nelson-Oppen combination in conjunction with the techniques presented in Section 5 and Section 6, because Nelson-Oppen method allows quantifier-free combinations of formulas that themselves need not be quantifier-free. For example, we can use Nelson-Oppen technique to decide quantifier-free combinations of two-variable logic with counting, $[\exists^*\forall^*]_=$ formulas, and linear arithmetic, thus combining the ideas from all three approaches.

Acknowledgements. We thank Darko Marinov for useful comments on an earlier version of this paper and useful discussions about the use of sets in symbolic execution. We thank Nguyen Huu Hai for useful remarks on an earlier version of this paper. We thank AIOOL'04 referees for useful feedback.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[2] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004: International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, March 2004.

[4] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[5] C. Boyapati, R. Lee, and M. C. Rinard. A type system for preventing data races and deadlocks. In *Proc. 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[6] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[7] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA'03*, 2003.

[8] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *CAV*, pages 355–367, 2003.

[9] E. Grädel. Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics. In *Proceedings of Kalmár Workshop on Logic and Computer Science, Szeged*, 2003.

[10] E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Proceedings of 12th IEEE Symposium on Logic in Computer Science LICS '97, Warschau*, 1997.

[11] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP*, 2004.

[12] D. Jackson. Alloy: a lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.

[13] D. Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980.

[14] V. Kuncak and M. Rinard. On role logic. Technical Report 925, MIT CSAIL, 2003.

[15] V. Kuncak and M. Rinard. Generalized records and spatial conjunction in role logic. In *11th Annual International Static Analysis Symposium (SAS'04)*, Verona, Italy, August 26–28 2004.

[16] V. Kuncak and M. Rinard. On the decision procedure for set-valued fields. Technical report, MIT CSAIL, November 2004.

[17] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[18] F. Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology And Software Technology (AMAST'2004)*, volume 3116 of *Lectures Notes in Computer Science*, pages 332–346. Springer-Verlag, July 2004.

[19] M. Mortimer. On languages with two variables. *Zeitschr. für math. Logik und Grundlagen der Math.*, 21:135–140, 1975.

[20] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[21] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[22] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proc. 12th ECOOP*, 1998.

[23] L. Pacholski, W. Szwast, and L. Tendera. Complexity results for first-order two-variable logic with counting. *SIAM J. on Computing*, 29(4):1083–1117, 2000.

[24] I. Pollet, B. L. Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of java programs. In *ECOOP*, 2001.

[25] I. Pratt-Hartmann. Complexity of the two-variable fragment with (binary-coded) counting quantifiers. *CoRR*, cs.LO/0411031, 2004.

[26] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 83–94. ACM Press, 2002.

[27] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (Volume 1).* Elsevier and The MIT Press, 2001.

[28] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS*, pages 29–37, 2001.

[29] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 281–293, 2000.

[30] T. Wies. Symbolic shape analysis. Master's thesis, Max-Planck Instutut für Informatik, Sep 2004.

[31] C. G. Zarba. Combining sets with elements. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 762–782. Springer, 2004.

[32] K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.