Symbolic Execution in CPAchecker

Thomas Lemberger
lemberge@fim.uni-passau.de

ABSTRACT

The value analysis of CPAchecker, a successful tool for configurable software verification, possesses high performance but is not able to handle non-deterministic values. This results in a high amount of falsely positive results. While counterexample-checks with other analyses are currently used to solve this problem, we extend the existing value analysis to allow the handling of non-deterministic values and introduce a new configurable program analysis (CPA) to track relations between such values. Our evaluation shows that this allows us to analyze test programs that use non-deterministic values correctly without the need for counterexample-checks.

1. MOTIVATION

With the rise of ubiquitious computing, reactive software systems that constantly receive input from the outside become more and more common. Because of their non-functional behaviour, the unreliability of testing becomes even more severe for such programs. In constrast to this, automatic software verification offers a more reliable alternative by analyzing all possible behaviours of a program. Configurable software verification[5] and its implementation CPAchecker[6] represent a recent approach to software verification that has been successful in multiple iterations of the Competition on Software Verification (SV-COMP) [1] [2] [3].

A prominent CPA used in CPAchecker is the value analysis CPA [7]. But while it offers high efficiency, it only tracks explicit values and cannot handle non-deterministic ones. Consider the simple example program in Listing 1 and its analysis by the value analysis CPA in Figure 1. It can easily be seen that the non-deterministic value cannot be handled and as such is discarded. Because of this over-approximation, necessary information about the relation between a and b gets lost and the safety property in Line 14 is seen as violated.

This characteristic results in a high amount of false positives. In practice, this problem is countered by the use of counterexample checks and strengthening through other CPAs. Instead of that, we will introduce symbolic values to the existing value analysis CPA to allow it tracking of non-deterministic values. In addition, we specify a new constraints CPA to track constraints on these symbolic values.

```
1  extern __nondet_int();
2
3  int main() {
4    int a = __nondet_int();
5    int b;
6
7    if (a >= 0) {
8      b = a;
9
10   } else {
11     b = a + 1;
12   }
13
14   if (b < a) {
15 ERROR:
16     return -1;
17   }
18 }
```

Listing 1: A simple non-deterministic program

First, we will give the formal specification of a composite CPA (which we will call symbolic execution CPA) using these two components. After this, an implementation of this CPA based on CPAchecker and its performance based on test sets of SV-COMP 2015 will be presented. We will show that the addition of symbolic value tracking allows the correct analysis of a large amount of programs priorly not possible through the value analysis CPA.

2. SPECIFICATION

The symbolic execution CPA is a composite CPA of the *location CPA* which tracks the current location in the syntactic structure of a program, the *symbolic value CPA* which tracks and computes the deterministic and non-deterministic values of variables and the *constraints CPA*, which tracks encountered assumptions in form of constraints for each location's variables. The formal definitions of composite CPA and location CPA are used as defined in [5].

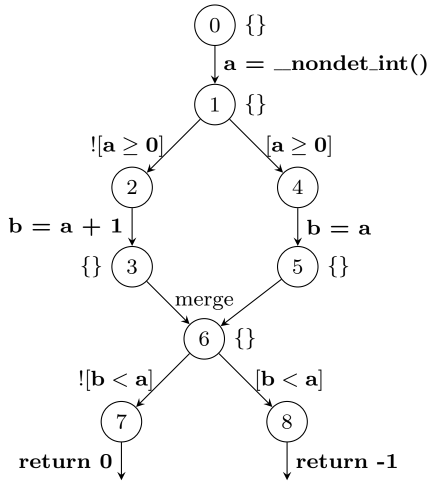

Figure 1: Analysis of the program in Listing 1 by the value analysis CPA. The current state of the CPA is noted after every assignment edge.

For the sake of formal simplicity, we assume that all values are integers and that a program only consists of assignments and assumptions.

2.1 Symbolic value CPA

The described CPA is an extension of the priorly mentioned value analysis CPA. Instead of using a bottom element $\perp$, we represent an unreachable state through the absence of a valid transfer. The symbolic value CPA is defined as

$$\mathbb{S} = (D_{\mathbb{S}}, \rightsquigarrow_{\mathbb{S}}, merge^{sep}\ stop^{sep})$$

with abstract domain $D_{\mathbb{S}}$, transfer relation $\rightsquigarrow_{\mathbb{S}}$, merge operator $merge^{sep}$ and stop operator $stop^{sep}$.

The abstract domain $D_{\mathbb{S}}$ is defined as $D_{\mathbb{S}} = (C, \mathcal{E}, [\![\cdot]\!])$ with C being the set of concrete program states, $\mathcal{E}$ the semi-lattice of possible abstract states and $[\![\cdot]\!]$ the concretization function.

$$\mathcal{E} = (V_{\mathbb{S}}, \sqsubseteq, \sqcup, v_{\top})$$

The elements of the semi-lattice are partial functions of $V_{\mathbb{S}} = X \rightarrow (\mathbb{Z} \cup \mathcal{Z}_{\mathbb{S}})$ mapping program variables in its definition range to concrete values of $\mathbb{Z}$ or to symbolic values $\mathcal{Z}_{\mathbb{S}} = S_I \cup S_E$. S_I describes all symbolic identifiers and S_E is the set of all valid symbolic expressions. Each expression is a symbolic expression if at least one symbolic identifier occurs in it. The definition range of a function f is defined as $\text{def}(f) = \{x \mid \exists y : f(x) = y\}$. The definition range of an abstract variable assignment of the type $V_{\mathbb{S}}$ consists of all program variables $x \in X$ whose value is known, with X being the set of all program variables occurring in the program. There are multiple reasons for unknown values: Uninitialized variables are unknown, but input values or calls to unknown functions and operations are, too.

To define $\sqsubseteq$, we first have to define the relation between concrete and abstract states. A concrete state $c \in C$ is *compliant* to an abstract variable assignment v, if an arbitrary, but valid assignment of concrete values to symbolic identifiers $d : S_I \rightarrow \mathbb{Z}$ exists, so that for all $x \in def(v)$ one of the following conditions is fulfilled: (1) the abstract assignment is a concrete value that equals the value in the concrete state, that means $c(x) = v(x)$, or (2) the abstract assignment $v(x)$ is a symbolic value that can be evaluated to $c(x)$ by replacing all occurring symbolic identifiers $i \in S_I$ with $d(i)$. The concretization function $[\![\cdot]\!]$ then assigns all compliant concrete states to an abstract state v.

For two abstract states $v, v' \in V_{\mathbb{S}}$, the concrete states that are covered by v are a subset of the ones covered by v', noted as $v \sqsubseteq v'$, if all of the following conditions hold: (1) v' must only contain value assignments also present in v, that is $\text{def}(v') \subseteq \text{def}(v)$, (2) $\forall x \in \text{def}(v') : v'(x) \in \mathbb{Z} \Rightarrow v'(x) = v(x)$ and (3) a bijective function alias : $S_I \rightarrow S_I$ exists that maps each symbolic identifier of S_I to another symbolic identifier, so that $\forall x \in \text{def}(v') : v'(x) \in \mathcal{Z}_{\mathbb{S}} \Rightarrow v(x)$ results from $v'(x)$ by replacing all $i \in S_I$ occurring in $v'(x)$ with $\text{alias}(i)$. Constraint 3 ensures that not the symbolic values themselves are compared, but their coverage of concrete states. Consider $v = \{(a, s1), (b, s1 + 5)\}$ and $v' = \{(a, s2), (b, s2 + 5)\}$ with $a, b \in X$ and $s1, s2 \in S_I$. As $s1$ and $s2$ represent any possible value, v represents the same amount of concrete states as v', that is all $c \in C$ with $c(b) = c(a) + 5$ and an arbitrary $c(a)$. Because of this, $v' \sqsubseteq v$ (and in this case even $v \sqsubseteq v'$).

Note that $v' \sqsubseteq v \Rightarrow [\![v']\!] \subseteq [\![v]\!]$, but $[\![v']\!] \subseteq [\![v]\!] \not\Rightarrow v' \sqsubseteq v$.

The join $\sqcup$ holds the least upper bound of two lattice elements $v, v' \in V_{\mathbb{S}}$ with

$(v \sqcup v')(x) = v(x)$ for all $x \in \text{def}(v) \cap \text{def}(v')$ and $v(x) = v'(x)$.

The top element of $\mathcal{E}$ is defined as $v_{\top} = \{\}$, i.e. no known assignments exist. It represents all possible concrete assignments, formally expressed as $[\![v_{\top}]\!] = C$.

The transfer relation $\rightsquigarrow_{\mathbb{S}}$ contains the transfer $v \overset{g}{\rightsquigarrow} v''$, if one of the following conditions is true:

1. $g = (l, assume(p), l')$, $\phi(p, v)$ is satisfiable and:

$$v''(x) = \begin{cases} c & \text{if } c \text{ only satisfying assignment for } \phi(p,v) \text{ and } x \notin \text{def}(v) \\ y & \text{if } x \notin \text{def}(v) \text{ and } x \text{ appears in } p.\ y \in S_I \text{ and } y \text{ is a new value that has not been used in any other state before} \\ v(x) & \text{otherwise} \end{cases}$$

   with

$$\phi(p, v) := p \wedge (\bigwedge_{\substack{x \in \text{def}(v), \\ v(x) \in \mathbb{Z}}} x = v(x))$$

   Note: ϕ performs an over-approximation, as only variables $x \in \text{def}(v)$ with an explicit value are considered.

2. $g = (l, w := e, l')$ and:

$$v''(x) = \begin{cases} \text{eval}(e, v') & \text{if } x = w \\ v'(x) & \text{if } x \in \text{def}(v) \text{ and } x \neq w \end{cases}$$

with

$$v'(x) = \begin{cases} y & \text{if } x \notin \text{def}(v) \text{ and } x \text{ appears in } e.\ y \in S_I \text{ and } y \text{ is a new value that has not been used in any other state before} \\ v(x) & \text{if } x \in \text{def}(v) \end{cases}$$

and $\text{eval}(e, v')$ defined as the evaluation of an expression e with the given assignment. If any symbolic value occurs in $v(e)$, it is only partially evaluated. In this case, $\text{eval}(e, v') \in \mathcal{Z}_{\mathbb{S}}$.

3. $v'' = v_\top$.

The merge operator merge^{sep} is defined as $\text{merge}^{sep}(e, e') = e'$. That means no merge is performed for different value states at the same location. The stop operator stop^{sep} considers every state of the reached set independently when checking for coverage. It is defined as $\text{stop}^{sep}(e, R) = \exists e' \in R : e \sqsubseteq e'$.

2.2 Constraints CPA

The constraints CPA is a CPA

$$\mathbb{C} = (D_{\mathbb{C}}, \rightsquigarrow_{\mathbb{C}}, merge^{sep}, stop^{sep})$$

that tracks constraints (i.e. boolean formulas) on variables that are created by assume edges. The abstract domain $D_{\mathbb{C}}$ is defined by

$$D_{\mathbb{C}} = (C, \mathcal{C}, [\![\cdot]\!])$$

with concrete states C, the semi-lattice $\mathcal{C}$ and concretization function $[\![\cdot]\!]$.

The abstract states described by $\mathcal{C} = (2^\gamma, \sqsubseteq, \sqcup, \top)$ consist of constraints of γ. γ contains all possible boolean expressions over $\mathbb{Z} \cup \mathcal{Z}_{\mathbb{S}}$, including symbolic expressions. An abstract state $a \in \mathcal{C}$ is interpreted as the conjunction of all its constraints.

For two given states $a, a' \in \mathcal{C}$, state a is lesser than or equal to a', if a bijective funtion alias : $S_I \rightarrow S_I$ exists, so that $a'' \subseteq a$ with a'' resulting from a' by replacing all symbolic identifiers $i \in S_I$ occurring in constraints of a' with alias(i).

The concretization function $[\![\cdot]\!]$ maps an abstract state to all concrete states that satisfy this abstract state's constraints.

$$[\![a]\!] = \{c \in C |\ c \vDash \bigwedge_{\varphi \in a} \varphi\}$$

Note: We assume that the empty conjunction represents true, that means $\bigwedge_{\varphi \in \{\}} \varphi = true$.

Constraint CPA's transfer relation $\rightsquigarrow_{\mathbb{C}}$ contains the transfer $a \overset{g}{\rightsquigarrow}_{\mathbb{C}} a'$ if one of the following conditions is fulfilled: (1) $g = (l, assume(p), l')$, $a' = a \cup p$ and a does not contain any variable $x \in X$ (by specifying that a must not contain any variable we enforce that variables are always replaced by concrete (explicit or symbolic) values through strengthening), or (2) $a' = a$ otherwise.

As p is only added as a constraint if no variable of X occurs in a, all resulting states of this transfer relation will be satisfiable, unless p is a contradiction ($a \neq a$, for instance). Because of this, we completely disregard checking for satisfiability of constraints here. Instead, we rely on satisfiability checks during strengthening by other CPAs. In the following we will use the priorly defined symbolic value CPA to replace newly added constraints' variables with values and check for the whole state's satisfiability.

2.3 Composition of CPAs

$\mathcal{C}_{\mathbb{LSC}} = (\mathbb{L}, \mathbb{S}, \mathbb{C}, \rightsquigarrow_x, merge^{sep}, stop^{sep})$ is the composite CPA of a location CPA and the priorly defined symbolic value CPA and constraints CPA. The transfer relation

$$\rightsquigarrow_x : D_{\mathbb{L}} \times D_{\mathbb{S}} \times D_{\mathbb{C}} \rightarrow D_{\mathbb{L}} \times D_{\mathbb{S}} \times D_{\mathbb{C}}$$

contains the transfer $(l, v, a) \overset{g}{\rightsquigarrow}_x (l', v', a'')$ if $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} l'$, $v \overset{g}{\rightsquigarrow}_{\mathbb{S}} v'$ and $a \overset{g}{\rightsquigarrow}_{\mathbb{C}} a'$ and if the strengthen operator $\downarrow_{\mathbb{C},\mathbb{S}}$ is defined for a' and v' with $\downarrow_{\mathbb{C},\mathbb{S}} (a', v') = a''$.

The strengthen operator $\downarrow_{\mathbb{C},\mathbb{S}}: \mathbb{C} \times \mathbb{S} \rightarrow \mathbb{C}$ uses the value assignment $v(x)$ of the given second operand to replace variables of the latest, unstrengthened constraint of the given constraints state. This way, meaningful constraints are created that show relations between symbolic values and can be checked for satisfiability.

$\downarrow_{\mathbb{C},\mathbb{S}} (a, v) = a'$ is defined if a' is the result of replacing all program variables $x \in X$ occurring in a with $v(x)$ and if $\bigwedge_{\varphi \in a'} \varphi$ is satisfiable.

An analysis of the previous test program using symbolic execution can be seen in Figure 2. By storing information about the relationship of variables a and b, the CPA can derive at location 6/6' that b can't be smaller than a.

3. EVALUATION

The specified CPA was implemented in CPAchecker[6] using the existing CompositeCPA and LocationCPA. The existing value analysis CPA was extended to fulfil the specification of our symbolic value CPA, while the constraints CPA was implemented as a new, own CPA. To check the satisfiability of constraints, an external SAT checker is used. For these benchmarks, MathSAT5 is used with the bitvector theory, encoding float values as floats. The analysis is part of CPAchecker's trunk since revision 16052 and can be activated by using the configuration `valueAnalysis-symbolic`. As development is ongoing, the current revision at the time of this writing, that is 16433, was chosen for benchmarks.

Benchmarks were performed on a subset of the SV-COMP 2015 test set, excluding sets CPAchecker's value analysis CPA has no competency in. These excluded sets are namely "Concurrency", "Memory Safety", "Recursive" and "Termination". An overview of all test sets can be found at [4]. Tests were executed on Intel Xeon E7-4870 machines at 2.40GHz with 322GB of available memory, a memory limit of 15.0GB per run and a time limit of 900 seconds. A Java heap memory limit of 10.5GB per run was used.

All benchmark results are available at `http://leostrakosch.github.io/symbolicValueAnalysis`.

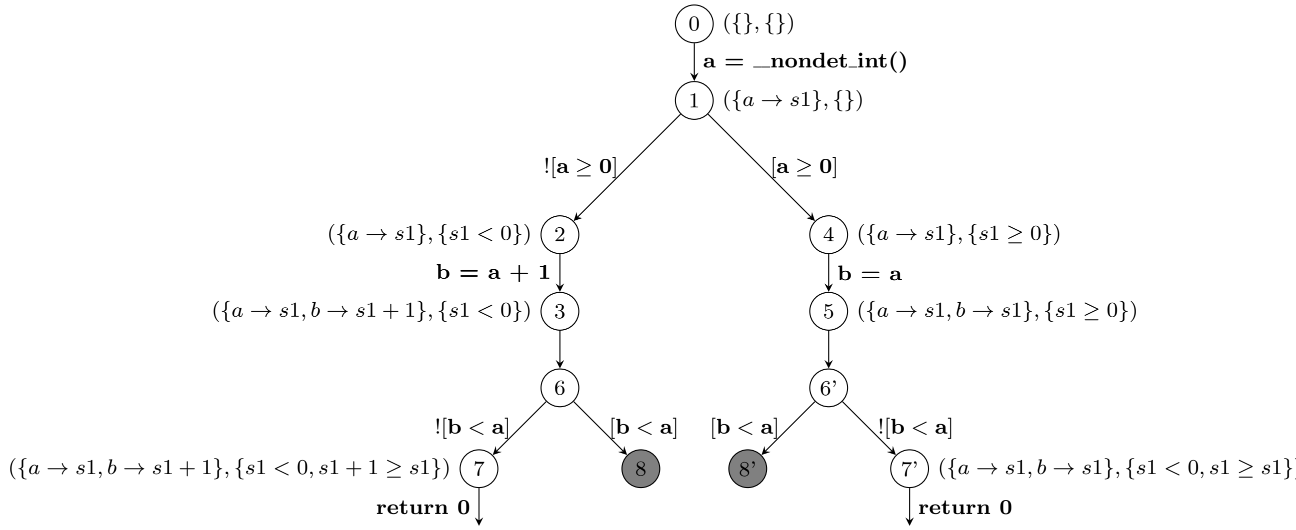

Figure 2: Analysis of the program in Listing 1 by the symbolic execution CPA. The locations reported as unreachable are tinted grey. The state is provided as a tuple at each relevant node, the first element being the state of $\mathbb{S}$, the second of $\mathbb{C}$. The state of the location CPA can be derived from the node's numbers.

Figure 3 shows the performance of the default value analysis CPA without counterexample-guided abstraction refinement (CEGAR [7]) or counterexample checks next to the performance of our symbolic execution configuration. Generally speaking, basic value analysis strongly outperforms symbolic execution in numbers due to symbolic execution taking too long. Lots of timeouts occur due to (a) path explosion, an exponential increase in possible paths based on the number of branches (that is if-statements and loops) as a result of the low level of abstraction of symbolic execution, and (b) the bad performance of SAT checks with a large number of variables and arithmetic operations like non-linear multiplication. Path explosion can easily be illustrated when looking at the algorithm statistics of a simple program using multiple if- and goto-statements.[1] While basic value analysis only needs 1241 iterations to find a possible property violation, with the biggest waitlist consisting of 42 entries, analysis with symbolic execution needs 14572 iterations with the biggest waitlist consisting of 543 states to find the same property violation. In addition to this enormous growth in iterations, SAT checks are responsible for up to 95.0% of CPU time in analyses of programs using non-deterministic values that are *not* specifically constructed for challenging SAT checkers.

Despite this reduced speed, symbolic execution's ability to track non-deterministic values allows it to get a correct result in 35 cases in which value analysis does not. In seven of these value analysis receives timeouts, and in 28 value analysis provides a false positive. Symbolic execution does never produce a false result when value analysis produces a correct one.

When using a timelimit of 1800 seconds, symbolic execution is able to compute 9 more results than with a timelimit of

	Value	SymEx	Overall
correct	1995	833	3038
true positives	632	246	802
true negatives	1363	587	2236
unique true positives	393	7	-
unique true negatives	804	28	-
false positives	271	52	-
unique false positives	222	3	-
false negatives	0	0	-
timeouts	662	1970	-
errors	110	182	-
memory errors	0	1	-

Figure 3: Result of runs of value analysis (Value) and symbolic execution (SymEx) on SV-COMP 2015 test sets. "true positive" represents a found property violation in a program containing a property violation, "true negative" represents no found violation in a program without violations.

[1] Namely program `ldv-validator-v0.6/linux-stable-9ec4f65-1-110_1a-drivers-rtc-rtc-tegra.ko-entry_point_false-unreach-call.cil.out.c`

	Bitvector theory	Integer theory
correct	833	815
false positives	52	93
false negatives	0	5
timeouts	1970	1944
errors	182	175
memory errors	1	1

Figure 4: Results of symbolic execution CPA using bitvector theory with floats and integer theory on SV-COMP 2015 test sets.

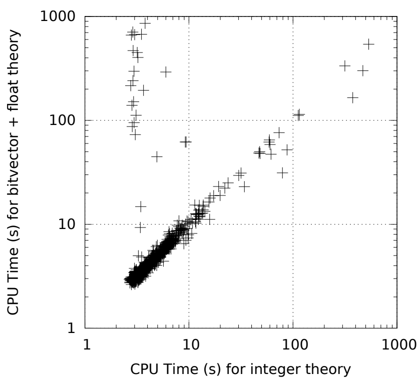

Figure 5: Runtime performance of symbolic execution using an integer theory in comparison to bitvector + float theory.

900 seconds, of which seven are correct.

Of these long-taking runs, three are the result of path explosion, while the other five only consist of few iterations, but contain expressions resulting in long-taking SAT checks, namely bitvector computations, non-deterministic floats and non-linear multiplications.

These performance issues can be improved by using another theory in SAT checks. By using an integer theory instead of bitvector, runtime can be greatly decreased in some cases, but a worse precision is achieved. It must be noted that the use of an integer theory results in an unsound analysis, also. Figure 3 shows the difference between the use of a bitvector theory with floats and the use of an integer theory. The scatter plot in Figure 3 shows the runtime of all test runs that did return a result and gave the same one in integer theory and bitvector theory.

4. CONCLUSION

By extending the existing value analysis CPA with the ability to track non-deterministic values and the introduction of the new ConstraintsCPA, we were able to drastically reduce the number of falsely positive analysis results of value analysis without the need for counterexample checks. In contrast to predicate analysis, we take advantage of the value analysis's high performance with explicit values and only create boolean formulas when non-deterministic values occur. As a downside, symbolic execution, as it is based on value analysis, is far less potent in handling pointers than predicate analysis. But since this is only a very basic implementation, a big amount of potential exists.

When choosing a SMT theory, it must be considered whether to trade performance for precision depending on the underlying program's use of floats and bitwise operations known. To decrease runtime without a loss of precision, the problem of path explosion should be tackled. A valid approach to this could be CEGAR, which is already implemented for the value analysis CPA. The introduction of compact symbolic execution as described in [8] could be another useful extension to allow more efficient handling of loops. By resolving the problem of path explosion, the number of iterations and as such also the number of SAT checks could be decreased, while keeping the high level of precision of symbolic execution. A preliminary implementation of CEGAR with symbolic execution already showed high potential. Refining this approach will be the main focus of future work.

5. REFERENCES

[1] Beyer, D. Second Competition on Software Verification (Summary of SV-COMP 2013). In *Proc. TACAS* (2013), vol. 7795 of *LNCS*, Springer, pp. 594–609.

[2] Beyer, D. Status Report on Software Verification (Competition Summary SV-COMP 2014). In *Proc. TACAS* (2014), vol. 8413 of *LNCS*, Springer, pp. 373–388.

[3] Beyer, D. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In *Proc. TACAS* (2015), LNCS, Springer. (To appear).

[4] Beyer, D. SV-COMP15: Benchmark Verification Tasks. `http://sv-comp.sosy-lab.org/2015/benchmarks.php`, 2015. Last checked: 03.31.2015.

[5] Beyer, D., Henzinger, T. a., and Théoduloz, G. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV* (2007), vol. 4590 of *LNCS*, Springer, pp. 504–518.

[6] Beyer, D., and Keremoglu, M. E. CPAchecker: A tool for configurable software verification. In *Proc. CAV* (2011), vol. 6806 of *LNCS*, Springer, pp. 184–190.

[7] Beyer, D., and Löwe, S. Explicit-State Software Model Checking Based on CEGAR and Interpolation. In *Proc. FASE* (2013), vol. 7793 of *LNCS*, Springer, pp. 146–162.

[8] Slaby, J., Strejček, J., and Trtík, M. Compact Symbolic Execution. In *Proc. ATVA* (2013), LNCS, Springer, pp. 193–207.