Commutativity Condition Refinement

Kshitij Bansal[1], Eric Koskinen[2], and Omer Tripp[2]

[1] New York University
[2] IBM TJ Watson Research Center

Abstract. We present a technique for automatically generating commutativity conditions from data-structure specifications. We observe that one can pose the commutativity question in a way that does not introduce additional quantifiers. We then iteratively refine an under-approximation of the commutativity (and non-commutativity) condition for two data-structure methods m and n into an increasingly precise version. Our algorithm terminates if/when the entire state space has been considered, and can be aborted at any time to obtain a partial, sound commutativity condition.
We have implemented our technique in a tool called Servois, which uses an SMT solver. We show that we can automatically generate symbolic commutativity conditions for a range of data structures including Set, HashTable, Accumulator, Counter, and Stack. This work has the potential to impact a variety of contexts, in particular, multicore software where it has been realized that commutativity is at the heart of increased concurrency.

1 Introduction

Problem. Recent decades have seen a variety of paradigms emerge for constructing programs that exploit the opportunity for concurrency in multicore architectures. Parallelizing compilers, speculative execution, transactional memory, and futures are all examples of such paradigms.

Reasoning about available concurrency at the granularity of concrete memory (read/write) accesses is overly conservative. Linearizable data structures, which by now are available in the standard libraries of all popular languages (e.g., C# and Java), enable a higher level of reasoning in terms of commutativity conditions. Indeed, commutativity of data-structure operations has been shown to be a key avenue to improved performance [2] or ease of verification [7,6]. As an example, consider a concurrent Hashtable implementation, such as the Java `ConcurrentHashMap.put` calls with different keys. They clearly commute: the order in which the two calls are made is irrelevant at the level of the data-structure, and they can thus execute in parallel. However, if successful, both calls would update the `size` field of the map, which — at the concrete level — manifests as conflicting updates. Commutativity conditions expose such parallelization opportunities that would otherwise be wasted. Existing work on commutativity, and its role in code parallelization, has focused mainly on verification and enforcement of commutativity conditions. A notable example of the former is the technique by Kim and Rinard [5], which accepts as input manually specified conditions, in the form of logical formulas, and verifies them using the Jahob system. More distantly related works include Kulkarni et al. [8], who observe that varying degrees of commutativity specification precision are useful, as well as works on program synthesis [11,13,12].

This work. We present the first technique for synthesis of commutativity conditions. Given the abstract representation of a data structure (with a linearizable implementation) with operations $m, n, \ldots$, we compute for every pair m/n a commutativity condition. That is, if the condition is satisfied, then m and n are guaranteed to commute.

We have implemented our technique in a new tool called Servois, which is built upon the SMT solver CVC4 [1]. We demonstrate that our technique is able to synthesize precise commutativity conditions for a representative set of data structures, which includes HashSet, HashTable, Accumulator, Counter, and Stack. Some of the conditions are nontrivial to specify by hand. Our technique leverages the SMT power of combining multiple theories while carefully avoiding the introduction of quantifiers via a transformation on the (potentially partial) specification to an equivalent total specification.

2 Overview

As our running example, consider the Set abstract data type (ADT), whose state consists of a single variable, S, that stores an *unordered* collection of *unique* elements. We focus on two operations: `contains(`x`)/bool` and `add(`y`)/bool` (returns `true` if data structure is modified). Clearly `add` and `contains` commute if they refer to different elements in the set. Another case, which is slightly more subtle, is when both `add` and `contains` refer to the same element e, and in the prestate $e \in S$. In this case, in both orders of execution `add` and `contains` leave the set unmodified and return false and true, respectively.

The algorithm we describe in this paper automatically produces a precise logical formula that captures this commutativity condition. It also generates the conditions under which the methods *do not* commute: $x = y \wedge x \notin S$. We explain the algorithm using our running example and, for reference, provide the psuedocode of the algorithm in Figure 1.

```
1  REFINE_n^m(H, P) {
2      if valid( H ⇒ m and n commute)
3        φ := φ ∨ H;
4      else if valid( H ⇒ m and n don't commute)
5        φ̂ := φ̂ ∨ H;
6      else
7        let χc, χnc = counterex. above (resp.) in
8        let p = choose(H, P, χc, χnc) in
9          REFINE_n^m(H ∧ p, P \ {p});
10         REFINE_n^m(H ∧ ¬p, P \ {p}); }
11 main {
12     φ := false;   φ̂ := false;
13     try { REFINE_n^m(true, P); }
14     catch (InterruptedException e) { skip; }
15     return(φ, φ̂); }
```

Fig. 1. Pseudocode description of core algorithm

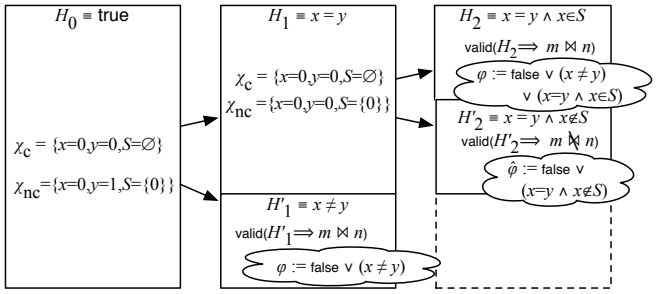

Fig. 2. Application of algorithm to the example

Iterative Refinement Algorithm. The main thrust of the algorithm is to recursively subdivide the state space via predicates until, at the base case, regions are found that are either entirely commutative or else entirely non-commutative. The conditions we incrementally generate are denoted φ and $\hat{\varphi}$, respectively. In the algorithm, we denote by H the logical formula that describes the current state space at a given recursive call. As expected, we begin with $H_0 = \mathsf{true}$. Our algorithm has three cases for a given H: (i) H describes a precondition for m and n in which m and n *always* commute; (ii) H describes a precondition for m and n in which m and n *never* commute; or (iii) neither of the above.

We illustrate how our algorithm proceeds on the running example in Figure 2. For these methods, true is visibly not a sound commutativity condition, as our algorithm determines via a *quantifier-free* (QF) *validity query* (described in more detail later) to an SMT solver. This returns the commutativity counterexample described above: $\chi_c = \{x = 0, y = 0, S = \emptyset\}$. true is also not a sufficient precondition for `add` and `contains` *not* to commute. We establish analogously, obtaining the non-commutativity counterexample $\chi_{nc} = \{x = 0, y = 1, S = \{0\}\}$.

Since $H_0 = \mathsf{true}$ is neither a commutativity nor a non-commutativity condition, we must refine H_0 into regions (or stronger conditions). In particular, we would like to perform a *useful* subdivision: Divide H_0 into an H_1 that allows χ_c but not χ_{nc}, and an H'_1 that allows χ_{nc} but not χ_c. To this end, the choose operation looks for a predicate p (from a suitable set of predicates $\mathcal{P}$, discussed later), such that $H_0 \wedge p \Leftarrow \chi_c$ while $H_0 \wedge \neg p \Leftarrow \chi_{nc}$ (or vice versa). The predicate $x = y$ satisfies this property. In the next two recursive calls, p is added as a conjunct to H, as shown in the second column of Figure 2: one with $H_1 \equiv \mathsf{true} \wedge x = y$ and one with $H'_1 \equiv \mathsf{true} \wedge x \neq y$.

Taking the H'_1 case, our algorithm makes another SMT query and finds that $x \neq y$ implies that `add` always commutes with `contains`. At this point, it can update the commutativity condition φ, letting $\varphi := \varphi \vee H'_1$, adding this H'_1 region to the growing disjunction. On the other hand, H_1 is neither a sufficient

commutativity nor a sufficient non-commutativity condition, and so our algorithm, again, produces the respective counterexamples: $\chi_c = \{x = 0, y = 0, S = \emptyset\}$ and $\chi_{nc} = \{x = 0, y = 0, S = \{0\}\}$. In this case, our algorithm selects the predicate $x \in S$, and makes two further recursive calls: one with $H_2 \equiv x = y \wedge x \in S$ and another with $H_2' \equiv x = y \wedge x \notin S$. In this case, it finds that H_2 is a sufficiently strong precondition for commutativity, while H_2' is a strong enough precondition for non-commutativity. Consequently, H_2 is added as a new conjunct to φ, yielding $\varphi \equiv x \neq y \vee (x = y \wedge x \in S)$. Similarly, $\hat{\varphi}$ is updated to be: $\hat{\varphi} \equiv (x = y \wedge x \notin S)$. No further recursive calls are made so the algorithm terminates, and we have obtained a precise (complete) commutativity/non-commutativity specification: $\varphi \vee \hat{\varphi}$ is valid.

Validity query. So far we relied on intutition for when `add` and `contains` commute. We make it this more precise now to give a flavor of the validity queries being generated, and to illustrate how we avoid quantifier alternation which arises when defining commutativity.

Let the abstract states be denoted by σ, σ', σ_m etc. (in our example, the abstract state was just the contents of the set). Denote by $\sigma \xrightarrow{m(a)/r} \sigma'$ the predicate which is true iff on application of method m with arguments a on state σ_0, the return value is r and new state is σ'. In our example, for `add` this predicate holds when $r = (a \notin S)$ and $S' = S \cup \{a\}$, and for `contains` this predicate holds when $r = (a \in S)$ and $S' = S$. We note that we only work with deterministic systems (this was not a limitation in our experiments), i.e. for any pre-state and method with specific arguments, there is atmost one post-state. At times, there might be no valid post state, and it is important to capture this. For example, one cannot pop an empty queue – so *push* followed by *pop* is possible, but *pop* as the first operation isn't.

Given the above definition, the definition of commutativity for a pair of methods m and n is: (i) for all abstract states σ_0, whenever m with arguments x (returning value r_m) followed by n with arguments y (returning value r_n) is possible, then the reverse application $n(y)$ followed by $m(x)$ is also possilbe and gives the same abstract state and return values (ii) vice-versa. In our notation, it would correspond to checking:

$$\forall \sigma_0, \sigma_1, \sigma_2, x, y, r_m, r_n.\ (\sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2 \implies (\exists \sigma_3.\sigma_0 \xrightarrow{n(y)/r_n} \sigma_3 \xrightarrow{m(x)/r_m} \sigma_2))$$
$$\wedge\ (\sigma_0 \xrightarrow{n(y)/r_n} \sigma_1 \xrightarrow{m(x)/r_m} \sigma_2 \implies (\exists \sigma_3.\sigma_0 \xrightarrow{m(x)/r_m} \sigma_3 \xrightarrow{n(y)/r_n} \sigma_2))$$

As one can see above, there is quantifier alternation if we use the natural encoding. When translated to a satisfiability query for an SMT solver, the inner existential quantifier stays as a universal quantifier. Since SMT solvers cannot handle quantifiers very well, we do a transformation which allows us to avoid quantifier alternation. We enforce that there is always a post-state by adding a new *Err* state in our set of abstract states. Whenever, there is no post-state for a given state, method and its arguments, we add a transition to the abstract *Err* state. Once in *Err* state, we always stay in *Err* state. Under this modified encoding, it is easy to prove that the following check encodes commutativity defined above:

$$\forall \sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4, x, y, r_m, r_n, r_m', r_n'.\ \sigma_0 \xrightarrow{m(x)/r_m} \sigma_1 \xrightarrow{n(y)/r_n} \sigma_2\ \wedge\ \sigma_0 \xrightarrow{n(y)/r_n'} \sigma_3 \xrightarrow{m(x)/r_m'} \sigma_4$$
$$\wedge\ ((\sigma_2 \neq Err \vee \sigma_4 \neq Err) \implies (r_m = r_m' \wedge r_n = r_n' \wedge \sigma_2 = \sigma_4))$$

Intuitively, with *exactly* one post state, the universal quantification works as well as the existential one to get a handle on the post state.

3 Evaluation

We have implemented the algorithm in Figure 1 in our tool Servois[‡]. We applied our tool to the SMT encoding of several data structures with states of the transition system encoded as tuples of variables and transitions (methods) encoded using logical formulae over old/new states, method arguments and return values. We experimented with two `choose` implementations. In the first, deubbed "simple", we only enforce

[‡] `http://cs.nyu.edu/~kshitij/projects/servois/`

Meth. m Meth.n	Simple	Poke	φ_n^m generated by Poke heuristic
Counter	Qs (time [s])	Qs (time [s])	
decrement ⋈ decrement	3 (0.11)	3 (0.11)	true
increment ▷ decrement	10 (0.36)	34 (0.91)	¬(0 = c)
decrement ▷ increment	3 (0.11)	3 (0.12)	true
decrement ⋈ reset	2 (0.10)	2 (0.10)	false
decrement ⋈ zero	6 (0.19)	26 (0.66)	¬(1 = c)
increment ⋈ increment	3 (0.12)	3 (0.11)	true
increment ⋈ reset	2 (0.09)	2 (0.10)	false
increment ⋈ zero	10 (0.30)	34 (0.86)	¬(0 = c)
reset ⋈ reset	3 (0.11)	3 (0.11)	true
reset ⋈ zero	9 (0.24)	30 (0.69)	0 = c
zero ⋈ zero	3 (0.11)	3 (0.11)	true
Accumulator			
increase ⋈ increase	3 (0.11)	3 (0.11)	true
increase ⋈ read	13 (0.31)	28 (0.63)	c + x1 = c
read ⋈ read	3 (0.09)	3 (0.09)	true
Set			
add ⋈ add	10 (0.40)	140 (4.47)	[y1 = x1 ∧ y1 ∈ S] ∨ [¬(y1 = x1)]
add ⋈ contains	10 (0.42)	122 (3.63)	[x1 ∈ S] ∨ [¬(x1 ∈ S) ∧ ¬(y1 = x1)]
add ⋈ getsize	6 (0.21)	31 (0.93)	x1 ∈ S
add ⋈ remove	6 (0.28)	66 (2.28)	¬(y1 = x1)
contains ⋈ contains	3 (0.18)	3 (0.16)	true
contains ⋈ getsize	3 (0.13)	3 (0.13)	true
contains ⋈ remove	17 (0.57)	160 (4.81)	[S\{x1} = {y1}] ∨ [¬(S\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ...] ∨ [...]
getsize ⋈ getsize	3 (0.12)	3 (0.13)	true
getsize ⋈ remove	13 (0.39)	37 (1.03)	¬(y1 ∈ S)
remove ⋈ remove	21 (0.75)	192 (6.47)	[S\{y1} = {x1}] ∨ [¬(S\{y1} = {x1}) ∧ y1 ∈ {x1} ∧ ...] ∨ [...]
HashTable			
get ⋈ get	3 (0.17)	3 (0.15)	true
get ⋈ haskey	3 (0.14)	3 (0.14)	true
put ▷ get	13 (0.47)	74 (2.37)	[H[x1=...] = H ∧ y1 ∈ keys] ∨ [¬(H[x1=...] = H) ∧ ¬(y1 = x1)]
get ▷ put	10 (0.37)	48 (1.54)	[H[y1] = y2] ∨ [¬(H[y1] = y2) ∧ ¬(y1 = x1)]
remove ▷ get	3 (0.17)	3 (0.16)	true
get ▷ remove	13 (0.45)	40 (1.23)	¬(y1 = x1)
get ⋈ size	3 (0.14)	3 (0.14)	true
haskey ⋈ haskey	3 (0.14)	3 (0.14)	true
haskey ⋈ put	10 (0.37)	52 (1.63)	[y1 ∈ keys] ∨ [¬(y1 ∈ keys) ∧ ¬(y1 = x1)]
haskey ⋈ remove	17 (0.59)	44 (1.36)	[x1 ∈ keys ∧ ¬(y1 = x1)] ∨ [¬(x1 ∈ keys)]
haskey ⋈ size	3 (0.14)	3 (0.14)	true
put ⋈ put	24 (0.97)	357 (13.50)	[H[y1] = y2 ∧ x2 = H[x1] ∧ ...] ∨ [H[y1] = y2 ∧ x2 = H[x1] ∧ ...] ∨ [...]
put ⋈ remove	6 (0.30)	33 (1.26)	¬(y1 = x1)
put ⋈ size	6 (0.29)	23 (0.82)	x1 ∈ keys
remove ⋈ remove	21 (0.89)	192 (6.95)	[keys\{x1} = {y1}] ∨ [¬(keys\{x1} = {y1}) ∧ y1 ∈ {x1} ∧ ...] ∨ [...]
remove ⋈ size	13 (0.45)	37 (1.13)	¬(x1 ∈ keys)
size ⋈ size	3 (0.14)	3 (0.14)	true
Stack			
clear ⋈ clear	3 (0.13)	3 (0.13)	true
clear ⋈ pop	2 (0.10)	2 (0.11)	false
clear ⋈ push	2 (0.12)	2 (0.11)	false
pop ⋈ pop	6 (0.23)	20 (0.62)	nextToTop = top
push ▷ pop	72 (2.14)	115 (3.53)	¬(0 = size) ∧ top = x1
pop ▷ push	34 (0.99)	76 (2.21)	y1 = top
push ⋈ push	13 (0.58)	20 (0.72)	y1 = x1

Fig. 3. Automatically generated commutativity conditions. The φ_n^m column shows the generated commutativity condition. When right moverness (▷) conditions are same for a pair of methods, we show them together in one row (⋈). **Qs** has the number of SMT queries made, and running time in seconds in parentheses. The experiments were run on a 2.53 GHz Intel Core 2 Duo machine with 8 GB RAM.

that the next predicate eliminates the counterexamples. Ties are broken by number of terms in predicate (those with lesser terms tried first) followed by order in which the predicate vocabulary is input. The second implementation, dubbed "poke", further recurses one level deeper to count how many predicates remain after its application that eliminate the counterexamples. The next predicate is the one leading to the lowest count. Ties are broken using the same strategy.

The results of our tool are given in Figure 3. On all of these examples our algorithm terminated without user interruption, thus these generated formulae are *precise* commutativity conditions. Although our algorithm is sound, we also manually validated implementation of Servois by examining its output with commutativity conditions manually written in prior works: [5] for Accumulator and Counter, [8] for Sets, and Dimitrov *et al.* [3] for HashTable.

Conclusion. Our work has given rise to a technique for automatically generating commutativity conditions of data-structures, and thus open for exploration a new application of SMT. By making use of recent advances (decision procedure for theory of finite sets) and a smart encoding of the commutativity constraints (avoiding quantifiers) we generate not only sound but also precise conditions for many data structures. These conditions can be derived statically and efficiently, and used in a variety of contexts including transactional boosting [4], open nested transactions [9], and other non-transactional concurrency paradigms such as race detection [3], automatic parallelization [10], etc.

References

1. Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)* (July 2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *Lecture Notes in Computer Science*, Springer, pp. 171–177. Snowbird, Utah.
2. Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comp. Sys. 32*, 4 (2015), 10.
3. Dimitrov, D., Raychev, V., Vechev, M., and Koskinen, E. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), Edinburgh, UK* (2014).
4. Herlihy, M., and Koskinen, E. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'08)* (2008).
5. Kim, D., and Rinard, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), ACM, pp. 528–541.
6. Koskinen, E., and Parkinson, M. J. The push/pull model of transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, Portland, OR, USA, June, 2015* (2015).
7. Koskinen, E., Parkinson, M. J., and Herlihy, M. Coarse-grained transactions. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010* (2010), M. V. Hermenegildo and J. Palsberg, Eds., ACM, pp. 19–30.
8. Kulkarni, M., Nguyen, D., Prountzos, D., Sui, X., and Pingali, K. Exploiting the commutativity lattice. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011* (2011), M. W. Hall and D. A. Padua, Eds., ACM, pp. 542–555.
9. Ni, Y., Menon, V., Adl-Tabatabai, A., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., and Shpeisman, T. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007* (2007), K. A. Yelick and J. M. Mellor-Crummey, Eds., ACM, pp. 68–78.
10. Rinard, M. C., and Diniz, P. C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 6 (November 1997), 942–991.
11. Solar-Lezama, A., Jones, C. G., and Bodík, R. Sketching concurrent data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (2008), pp. 136–148.
12. Vechev, M. T., and Yahav, E. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (2008), pp. 125–135.
13. Vechev, M. T., Yahav, E., and Yorsh, G. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010* (2010), pp. 327–338.