LEAP: A Tool for the Parametrized Verification of Concurrent Datatypes*

Alejandro Sánchez[1] and César Sánchez[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Institute for Information Security, CSIC, Spain

Abstract. This tool paper describes LEAP, a tool for the verification of concurrent datatypes and parametrized systems composed by an unbounded number of threads that manipulate infinite data[3].
LEAP receives as input a concurrent program description and a specification and automatically generates a finite set of verification conditions which are then discharged to specialized decision procedures. The validity of all discharged verification conditions implies that the program executed by any number of threads satisfies the specification. Currently, LEAP includes not only decision procedures for integers and Booleans, but it also implements specific theories for heap memory layouts such as linked-lists and skiplists.

1 Introduction

The target application motivating the development of LEAP is the verification of concurrent datatypes [16]. Concurrent datatypes are designed to exploit the parallelism of multiprocessor architectures by employing very weak forms of synchronization, like lock-freedom and fine-grain locking, allowing multiple threads to concurrently access the underlying data. The formal verification of these concurrent programs is a very challenging task, particularly considering that they manipulate complex data structures capable of storing unbounded data, and are executed by an unbounded number of threads.

The problem of verifying parametrized finite state systems has received a lot of attention in recent years. In general, the problem is undecidable [3]. There are two general ways to overcome this limitation: (*i*) algorithmic approaches, which are necessarily incomplete; and (*ii*) deductive proof methods. Typically, algorithmic methods—in order to regain decidability—are restricted to finite state processes [8, 9, 14] and finite state shared data. LEAP follows an alternative approach, by extending temporal deductive methods like Manna-Pnueli [20] with specialized proof rules for parametrized systems, thus sacrificing full automation to handle complex concurrency and data manipulation. Our target with LEAP is wide applicability, while improving automation is an important secondary goal.

* This work was funded in part by Spanish MINECO Project "TIN2012-39391-C04-01 STRONGSOFT"

[3] LEAP is under development at the IMDEA Software Institute. All examples and code can be downloaded from `http://software.imdea.org/leap`

Most algorithmic approaches to parametrized verification abstract both control and data altogether [1,2,21] reducing the safety to a (non)reachability problem in a decidable domain. In these approaches, data manipulation and control flow are handled altogether, and the verification is limited to simple theories such as Booleans and linear arithmetic. LEAP, on the other hand, separates the two concerns: (*i*) the concurrent interaction between threads; and (*ii*) the data being manipulated. The first concern is tackled with specialized deductive parametrized proof rules, which, starting from a parametrized system and a temporal specification, generate a finite number of verification conditions (VCs). The second aspect is delegated to decision procedures (DP) specifically designed for each datatype, which can prove the validity of VCs automatically. Our proof rules are designed to generate *quantifier free* VCs, for which it is much easier to design decidable theories and obtain automatic decision procedures.

There exists a wide range of tools for verifying concurrent systems. Smallfoot [4] is an automatic verifier that uses concurrent separation logic for verifying sequential and concurrent programs. Smallfoot depends on built-in rules for the datatypes, which are typically recursive definitions in separation logic. Unlike LEAP, Smallfoot cannot handle programs without strict separation (like shared readers) or algorithms that do not follow the unrolling that is explicit in the recursive definitions. TLA+ [7] is able to verify temporal properties of concurrent systems with the aid of theorem provers and SMT solvers, but TLA+ does not support decision procedures for data in the heap. Similarly, HAVOC [11] is capable of verifying C programs relying on Boogie as intermediate language and Z3 as backend. Neither Frama-C [12] nor Jahob [17] handle parametrized verification, which is necessary to verify concurrent datatypes (for any number of threads). The closest system to LEAP is STeP [19], but STeP only handled temporal proofs for simple datatypes. Unlike LEAP, none of these tools can reason about parametrized systems.

Chalice [18] is an experimental language that explores specification and verification of concurrency in programs with dynamic thread creation, and locks. VeriCool [28] uses dynamic framing (as Chalice does) to tackle the verification of concurrent programs using Z3 as backend. However, none of these tools implement specialized DPs for complex theories of datatypes. VCC [10] is an industrial-strength verification environment for low-level concurrent system code written in C. Despite being powerful, in comparison to LEAP it requires a great amount of program annotation.

So far, the current version of LEAP only handles safety properties, but support for liveness properties is ongoing work.

2 Formal Verification Using LEAP

Fig. 1 shows the structure of LEAP. LEAP receives as input a program and a specification. Fig. 2 presents an example of a procedure for inserting an element into a concurrent lock-coupling single-linked list. The input language is a C-like language with support for assignments—including pointers—, conditionals,

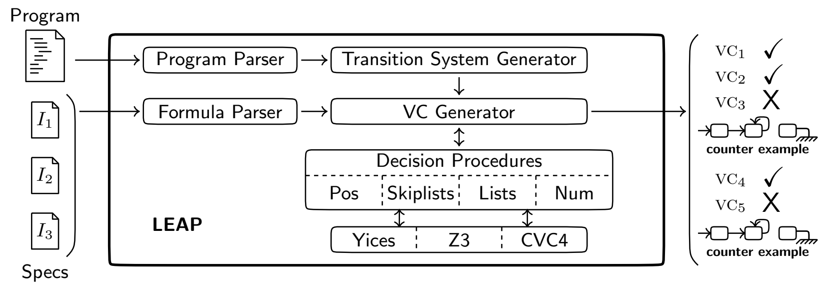

Fig. 1. Scheme of LEAP

loops and non-recursive function calls. Program lines can be assigned a label to refer to them later in a specification (e.g., `connect` labels line 15 in Fig. 2). In a specification both program lines and labels can be used to refer to an specific section of the program. The input language also supports atomic sections and ghost code. Ghost code is written between `$` and, is added only for verification purposes, and it is removed during compilation. Fig. 2 declares a global ghost variable `region` for keeping track of address of nodes belonging to the list, which is updated at line 15 when a new node is connected added to the list. Leap requires only small annotations of extra ghost. A specification consists of quantifier-free parametrized formulas describing the property to be verified. Consider the following specification, parametrized by thread id `i`:

```
    global
      addr head, tail
      ghost addrSet region
    procedure insert (e:elem)
      addr prev, curr, aux
    begin
 1: prev := head;
 2: prev->lock;
 3: curr := prev->next;
 4: curr->lock;
 5: while curr->data < e do
 6:    aux := prev;
 7:    prev := curr;
 8:    aux->unlock;
 9:    curr := curr->next;
10:    curr->lock;
11: end while
12: if curr != null /\curr->data > e then
13:    aux := malloc(e,null,#);
14:    aux->next := curr;
    :connect
15:    prev->next := aux
        $region := region Union {aux};$
16: end if
17: prev->unlock;
18: curr->unlock;
19: return
    end procedure
```

Fig. 2. Example of input program

```
vars: tid i
specification [aux_ready] :
  @connect(i). ->
    (rd(heap, prev(i)).data < e /\ rd(heap, curr(i)).data > e /\
     rd(heap, aux(i)).data = e /\
     rd(heap, prev(i)).next = curr(i) /\ rd(heap, aux(i)).next = curr(i))
```

This formula describes conditions that every thread `i` satisfy during insertion and that guarantee the preservation of the list shape when connecting the node pointed by `aux` to the list. In particular, `aux_ready` states that: (1) the node pointed by `prev` (resp. `curr`) stores a value lower (resp. higher) than `e`, (2) the node pointed by `aux` stores value `e`, and (3) the field `next` of the nodes pointed by `prev` and `aux` points to `curr`. We now give a brief description of how Leap generates the VCs starting from the program and the specifications received as input.

Verification Condition Generation. Given an input program P, LEAP internally creates an implicit representation of a parametrized transition system $S[M] = P_1 \| P_2 \| \ldots \| P_M$, where each P_j is an instance of program P. For example, if we consider the program from Fig. 2, $S[1]$ is the instance of $S[M]$ consisting of in a single thread running `insert` in isolation, and $S[2]$ is the instance of $S[M]$ consisting of two threads running `insert` concurrently. LEAP solves the uniform verification problem showing that all instances of the parametrized system satisfy the safety property by using specialized proof rules [25] which generate a finite number of VCs.

Each VC describes a small-step in the execution. All VCs generated by LEAP are quantifier free as long as the specification is quantifier free. We use the theory of arrays [5] to encode the local variables of a system with an arbitrary number of threads, but the dependencies with arrays are eliminated by symmetry. VCs are discharged to specialized DPs which automatically decide their validity. If all VCs are proved valid, then the specification is verified to be an invariant of the parametrized program. If a VC is not valid, then the DP generates a counter-model corresponding to an offending small-step of the system that leads to a violation of the specification. This is typically a very small heap snippet that the programmer can use to either identify a bug or instrument the program with intermediate invariants. Consider property `is_list` which states the list shape property, including that the ghost variable `region` stores the set of addresses of all nodes belonging to the list. Property `aux_ready` is not enough to prove `is_list` invariant. Fig. 3 shows a counter example returned by the decision procedure. The output can be used by the user as hint to strengthen `aux_ready`, in this case, indicating that `prev` must belong to `region` before executing line 15 of `insert`.

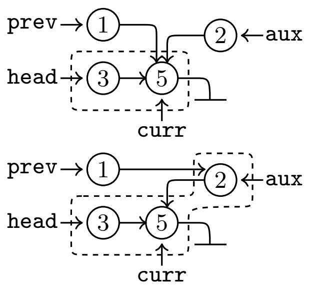

Fig. 3. A counter-example for `is_list` when executing line 15 (up shows before, down shows after). Dashed box represents `region`.

Decision Procedures. LEAP implements specialized decision procedures including some theories of heap memory layouts and locks [23, 24, 27] whose decidability is based on finite model theorems. Our implementation transforms each VC into queries to the corresponding DP. The decision procedures are implemented on top of off-the-shelf SMT solvers [13, 15]. LEAP currently includes decision procedures for Presburger arithmetic with finite sets and minimum, lock-based concurrent single-linked lists [23], concurrent skiplists of bounded height [24] and skiplists of arbitrary height [27]. The modular design of LEAP makes it straightforward to implement extensions for new program statements, theories and DPs.

Proof Graphs and Tactics. Proofs in LEAP are structured as *proof graphs*, which describes the inter-dependency between invariants. Proof graphs improve the efficiency of proof development and proof checking, by establishing the necessary support for proving consecution (see [20] and optionally specifying tactics and heuristics. Current implemented tactics include the use of simpler DPs with some symbols uninterpreted, lazy instantiation of supporting invariants, and applications of typical first-order tactics like equality propagation and removal of irrelevant literals. Tactics are very useful when performed prior to the discharge of a VC to the SMT solver, as bound sizes of candidate models are reduced. For instance, the proof graph for `is_list` includes:

```
=> is_list [15:aux_ready] { pruning : split-goal | | | simplify-pc}
```

indicating that in order to prove consecution for `is_list` at line 15, `aux_ready` is a useful support. The annotation `pruning` establishes a tighter domain bound calculation for the list DP. The graph also lists tactics `split-goal` and `simplify-pc`. In proof creation, these tactics can be explored automatically in parallel dumping the fastest option to the proof graph file for efficient proof checking.

3 Empirical Evaluation

Fig. 4 reports the use of Leap to verify some concurrent and sequential programs, executed on a computer with a 2.8 GHz processor and 8GB of memory. Each row includes the outcome of the verification of a single invariant. Rows 1 to 12 correspond to the verification of a concurrent lock-coupling single-linked lists implementing a set, including both shape preservation and functional properties. Formulas list and order state that the shape is that of an ordered single-linked list, lock describes the fine-grain lock ownership, next captures the relative position of local pointer variables, region constraints the region of the heap to contain precisely the list nodes and disj encodes the separation of new cells allocated by different threads. Functional properties include funSchLinear: search returns whether the element is present at the linearization point; funSchInsert and funSchRemove: a search is successful precisely when the element was inserted and not removed after; funRemove, funInsert and funSearch describe a scenario in which a thread manipulates different elements than all other threads: an element is found if and only if it is in the list, an element is not present after removal, and an element is present after insertion. Rows 13 to 16, and 17 to 20 describe the verification of two sequential implementation of a skiplist. The first implementation limits the maximum height to 3 levels, and the second considers an implementation in which the height can grow beyond any bound, using a more sophisticated DP. Rows 21 to 23 correspond to a parametrized ticket based mutual exclusion protocol. This protocol is infinite state using integers as tickets. Lines 24 to 26 correspond to a similar protocol that uses sets of integers.

The first four columns show (1) the formula's index (i.e., the number of threads parametrizing the formula), (2) the number of VCs discharged, (3) the number of such VCs proved by program location reasoning, and (4) by using a specialized DP. In all cases, all VCs are automatically verified. The next two columns report the total running time of discharging and proving all VCs (5) without using any tactic and (6) with tactics enabled. The next columns present the slowest (7) and average (8) running time to solve VCs, and the final column

		formula		#solved vc		Brute	Heurist.	DP time		LEAP
		idx	#vc	pos	dp	time	time	slowest	average	time
1	list	0	61	38	23	∞	18.67	11.90	0.30	0.20
2	order	1	121	62	59	998.35	1.12	0.03	0.01	0.47
3	lock	1	121	76	45	778.15	0.47	0.02	0.01	0.18
4	next	1	121	60	61	∞	2.11	0.61	0.01	0.59
5	region	1	121	95	26	∞	22.58	18.17	0.18	0.23
6	disj	2	181	177	4	121.74	0.19	0.01	0.01	0.12
7	funSchLinear	1	121	97	24	∞	6.29	3.04	0.05	0.08
8	funSchInsert	1	121	93	28	∞	4.15	1.91	0.03	0.08
9	funSchRemove	1	121	93	28	∞	5.40	2.60	0.04	0.10
10	funSearch	1	208	198	10	∞	3.54	1.57	0.01	0.34
11	funInsert	1	208	200	8	∞	0.50	0.01	0.01	0.22
12	funRemove	1	208	200	8	∞	1.41	0.95	0.01	0.24
13	skiplist$_3$	0	154	92	62	∞	1221.97	776.45	15.27	0.45
14	region$_3$	0	124	97	27	∞	27.50	17.36	0.34	0.58
15	next$_3$	0	84	65	19	∞	0.67	0.09	0.01	0.20
16	order$_3$	0	84	59	25	∞	9.66	7.80	0.10	1.31
17	skiplist	0	560	532	28	∞	19.79	5.40	0.24	0.15
18	region	0	1583	1527	56	∞	44.28	22.66	0.54	1.35
19	next	0	1899	1869	30	∞	3.19	0.32	0.02	1.59
20	order	0	2531	2474	57	∞	11.19	2.35	0.84	6.75
21	mutex	2	28	26	2	0.32	0.01	0.01	0.01	0.01
22	minticket	1	19	18	1	0.04	0.01	0.01	0.01	0.01
23	notsame	2	28	26	2	0.13	0.03	0.01	0.01	0.01
24	mutexS	2	28	26	2	0.44	0.04	0.01	0.01	0.01
25	minticketS	1	19	18	1	0.31	0.01	0.01	0.01	0.01
26	notsameS	2	28	26	2	0.14	0.02	0.01	0.01	0.01

Fig. 4. Verification running times (in secs.). ∞ represents a timeout of 30 minutes.

includes the analysis time without considering the running time of decision procedures. Our results indicate that LEAP can verify sophisticated concurrent programs and protocols with relatively small human intervention. Required annotation for our examples was around 15% of the source code (roughly 1 invariant—containing 6 primitive predicates each—every 7 lines). The time employed by LEAP to analyze the program and generate all VCs is a negligible part of the total running time, which suggests that research in DP design and implementation is the crucial bottleneck for scalability. Also, in practice, tactics are important for efficiency to handle non-trivial systems.

4 Future Work

We are considering the use of CIL/Frama-C as a front-end for C. Extending LEAP with support for liveness properties is ongoing work. Our approach consists of specializing generalized verification diagrams [6] and transition invariants [22] for parametrized systems. The development of new theories and decision procedures for new datatypes such as hash-maps and lock-free lists is currently under development. We are also exploring the possibility of increasing automation by automatically generating intermediate specifications. Our approaches include (1) how to apply effectively weakest precondition propagation for parametrized systems, and (2) extending our previous work on abstract interpretation-based invariant generation for parametrized systems [26] to handle complex heap layouts.

References

1. P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *Proc. of CAV'99*, vol. 1633 of *LNCS*, pages 134–145. Springer, 1999.
2. P. A. Abdulla, G. Delzanno, and A. Rezine. Approximated parameterized verif. of infinite-state processes with global conditions. *FMSD*, 34(2):126–156, 2009.
3. K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.
4. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proc. of FMCO'05*, vol. 4111 of *LNCS*, pages 115–137, 2005.
5. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. of VMCAI'06*, vol. 3855 of *LNCS*, pages 427–442. Springer, 2006.
6. A. Browne, Z. Manna, and H. B. Sipma. Generalized verification diagrams. In *Proc. of FSTTCS'95*, vol. 1206 of *LNCS*, pages 484–498. Springer, 1995.
7. K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. The TLA$^+$ proof system: Building a heterogeneous verification platform. In *Proc. of ICTAC'10*, vol. 6255 of *LNCS*, 2010.
8. E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proc. of PODC'87*, pages 294–303. ACM, 1987.
9. E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state procs. In *Proc. of PODC'86*, pages 240–248. ACM, 1986.
10. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. of TPHOLs'09*, vol. 5674 of *LNCS*, pages 23–42, 2009.
11. J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Proc. of POPL'09*, pages 302–314. ACM Press, 2009.
12. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C – a software analysis perspective. In *Proc. of SEFM'12*, vol. 7504 of *LNCS*, pages 233–247. Springer, 2012.
13. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, vol. 4963 of *LNCS*, pages 337–340. Springer, 2008.
14. E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Proc. of CADE'00*, vol. 1831 of *LNAI*, pages 236–254. Springer, 2000.
15. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Proc. of CAV'04*, vol. 3114 of *LNCS*, pages 175–188. Springer, 2004.
16. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan-Kaufmann, 2008.
17. V. Kuncak and M. C. Rinard. An overview of the Jahob analysis system: project goals and current status. In *Proc. of IPDPS'06*, IEEE Computer Society Press, 2006.

18. K. R. M. Leino. Verifying concurrent programs with Chalice. In *Proc. of VMCAI'10*, vol. 5944 of *LNCS*. Springer, 2010.
19. Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. de Alfaro, H. Devarajan, H. Sipma, and T. Uribe. STeP: The stanford temporal prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.
20. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer, 1995.
21. G. D. Marco Bozzano. Beyond parameterized verification. In *Proc. of TACAS'02*, vol. 2280 of *LNCS*, pages 221–235. Springer, 2002.
22. A. Podelsky and A. Rybalchenko. Transition invariants. In *Proc. of LICS'2004*, pages 32–41. IEEE Computer Society Press, 2004.
23. A. Sánchez and C. Sánchez. Decision procedure for the temporal verification of concurrent lists. In *Proc. of ICFEM'10*, vol. 6447 of *LNCS*, pages 74–89. Springer, 2010.
24. A. Sánchez and C. Sánchez. A theory of skiplists with applications to the verification of concurrent datatypes. In *NFM'11*, vol. 6617 of *LNCS*, pages 343–358. Springer, 2011.
25. A. Sánchez and C. Sánchez. Parametrized invariance for infinite state processes. *CoRR*, abs/1312.4043, 2013.
26. A. Sánchez, S. Sankaranarayanan, C. Sánchez, and B.-Y. E. Chang. Invariant generation for parametrized systems using self-reflection. In *Proc. of SAS'12*, vol. 7460 of *LNCS*, pages 146–163. Springer, 2012.
27. C. Sánchez and A. Sánchez. A decidable theory of skiplists of unbounded size and arbitrary height. *CoRR*, abs/1301.4372, 2013.
28. J. Smans, B. Jacobs, and F. Piessens. Vericool: An automatic verifier for a concurrent object-oriented language. In *Proc. of FMOODS'08*, vol. 5051 of *LNCS*, pages 220–239, 2008.