SACO: Static Analyzer for Concurrent Objects

E. Albert[1], P. Arenas[1], A. Flores-Montoya[2], S. Genaim[1],
M. Gómez-Zamalloa[1], E. Martin-Martin[1], G. Puebla[3], and G. Román-Díez[3]

[1] Complutense University of Madrid (UCM), Spain
[2] Technische Universität Darmstadt (TUD), Germany
[3] Technical University of Madrid (UPM), Spain

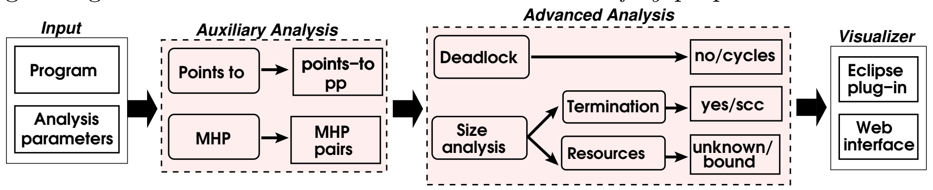
Abstract. We present the main concepts, usage and implementation of SACO, a static analyzer for concurrent objects. Interestingly, SACO is able to infer both *liveness* (namely termination and resource boundedness) and *safety* properties (namely deadlock freedom) of programs based on concurrent objects. The system integrates auxiliary analyses such as *points-to* and *may-happen-in-parallel*, which are essential for increasing the accuracy of the aforementioned more complex properties. SACO provides accurate information about the dependencies which may introduce deadlocks, loops whose termination is not guaranteed, and upper bounds on the resource consumption of methods.

1 Introduction

With the trend of parallel systems, and the emergence of multi-core computing, the construction of tools that help analyzing and verifying the behaviour of concurrent programs has become fundamental. Concurrent programs contain several processes that work together to perform a task and communicate with each other. Communication can be programmed using shared variables or message passing. When shared variables are used, one process writes into a variable that is read by another; when message passing is used, one process sends a message that is received by another. Shared memory communication is typically implemented using low-level concurrency and synchronization primitives These programs are in general more difficult to write, debug and analyze, while its main advantage is efficiency. The message passing model uses higher-level concurrency constructs that help in producing concurrent applications in a less error-prone way and also more modularly. Message passing is the essence of actors [1], the concurrency model used in *concurrent objects* [9], in Erlang, and in Scala.

This paper presents the SACO system, a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects. Essentially, each concurrent object is a monitor and allows at most one *active* task to execute within the object. Scheduling among the tasks of an object is cooperative, or non-preemptive, such that the active task has to release the object lock explicitly (using the **await** instruction). Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and start executing. When the result of a call is required by the caller to continue executing, the caller and the callee methods can be synchronized by means of *future variables*, which act as proxies for results initially unknown, as their computations are still incomplete.

The figure below overviews the main components of SACO, whose distinguishing feature is that it infers both *liveness* and *safety* properties.



SACO receives as input a program and a selection of the analysis parameters. Then it performs two auxiliary analyses: points-to and may-happen-in-parallel (MHP), which are used for inferring the more complex properties in the next phase. As regards to liveness, we infer termination as well as resource boundedness, i.e., find upper bounds on the resource consumption of methods. Both analyses require the inference of *size relations*, which are gathered in a previous step. Regarding *safety*, we infer deadlock freedom, i.e., there is no state in which a non-empty set of tasks cannot progress because all tasks are waiting for the termination of other tasks in the set, or otherwise we show the tasks involved in a potential deadlock set. Finally, SACO can be used from a command line interface, a web interface, and an Eclipse plugin. It can be downloaded and/or used online from its website http://costa.ls.fi.upm.es/saco.

2 Auxiliary Analyses

We describe the auxiliary analyses used in SACO by means of the example below:

```
1  class PrettyPrinter{
2   void showIncome(Int n){...}
3   void showCoin(){...}
4  }//end class
5  class VendingMachine{
6  Int coins;
7  PrettyPrinter out;
8   void insertCoins(Int n){
9    Fut⟨void⟩ f;
10   while (n>0){
11      n=n−1;
12      f=this ! insertCoin();
13      await f?; }
14 }
15  void insertCoin(){
16   coins=coins+1;
17 }
18  Int retrieveCoins(){
19   Fut⟨void⟩ f;
20   Int total=0;
21   while (coins>0){
22    coins=coins−1;
23    f=out ! showCoin();
24    await f?;
25    total=total+1; }
26   return total;
27  }
28 }//end class
29 //main method
30 main(Int n){
31   PrettyPrinter p;
32   VendingMachine v;
33   Fut⟨Int⟩ f;
34   p=new PrettyPrinter();
35   v=new VendingMachine(0,p);
36   v ! insertCoins(n);
37   f=v ! retrieveCoins();
38   await f?;
39   Int total=f.get;
40   p!showIncome(total);
41 }
```

We have a class PrettyPrinter to display some information and a class VendingMachine with methods to insert a number of coins and to retrieve all coins. The main method is executing on the object This, which is the initial object, and receives as parameter the number of coins to be inserted. Besides This, two other concurrent objects are created at Line 34 (L34 for short) and L35. Objects can be seen as buffers in which tasks are posted and that execute in parallel. In particular, two tasks are posted at L36 and L37 on object v. insertCoins executes asynchronously on v. However, the **await** at L38 synchronizes the execution of This with the completion of the task retrieveCoins in v by means of the future variable f. Namely, at the **await**, if the task spawned at L37 has not finished, the processor is released and any available task on the This object could take it. The result of the execution of retrieveCoins is obtained by means of the blocking **get** instruction which *blocks* the execution of This until the future variable f is ready. In general, the use of **get** can introduce deadlocks. In this case, the **await** at L38 ensures that retrieveCoins has finished and thus the execution will not block.

Points-to Analysis. Inferring the set of memory locations to which a reference variable *may* point-to is a classical analysis in object-oriented languages. In SACO we follow Milonava et al. [11] and abstract objects by the *sequence of allocation sites* of all objects that lead to its creation. E.g., if we create an object o_1 at program point pp_1, and afterwards call a method of o_1 that creates an object o_2 at program point pp_2, then the abstract representation of o_2 is $pp_1.pp_2$. In order to ensure termination of the inference process, the analysis is parametrized by k, the maximal length of these sequences. In the example, for any $k \geq 2$, assuming that the allocation site of the This object is ϵ, the points-to analysis abstracts v and out to $\epsilon.35$ and $\epsilon.34$, respectively. For $k = 1$, they would be abstracted to 35 and 34. As variables can be reused, the information that the analysis gives is specified at the program point level. Basically, the analysis results are defined by a function $\mathcal{P}(o_p, pp, v)$ which for a given (abstract) object o_p, a program point pp and a variable v, it returns the set of abstract objects to which v may point to. For instance, $\mathcal{P}(\epsilon, 36, v) = 35$ should be read as: when executing This and instruction L36 is reached, variable v points to an object whose allocation site is 35. Besides, we can trivially use the analysis results to find out to which task a future variable f is pointing to. I.e., $\mathcal{P}(o_p, pp, f) = o.m$ where o is an abstract object and m a method name, e.g., $\mathcal{P}(\epsilon, 37, f) = 35.\mathsf{retrieveCoins}$. Points-to analysis allows making any analysis object-sensitive [11]. In addition, in SACO we use it: (1) in the resource analysis in order to know to which object the cost must be attributed, and (2) in the deadlock analysis, where the abstraction of future variables above is used to spot dependencies among tasks.

May-Happen-in-Parallel. An MHP analysis [10, 3] provides a safe approximation of the set of pairs of statements that can execute in parallel across several objects, or in an interleaved way within an object. MHP allows ensuring absence of data races, i.e., that several objects access the same data in parallel and at least one of them modifies such data. Also, it is crucial for improving the accuracy of deadlock, termination and resource analysis. The MHP analysis implemented in SACO [3] can be understood as a function $\mathcal{MHP}(o_p, pp)$ which returns the set of program points that may happen in parallel with pp when executing in the abstract object o_p. A remarkable feature of our analysis is that it performs a local analysis of methods followed by a composition of the local results, and it has a polynomial complexity. In our example, SACO infers that the execution of showIncome (L2) cannot happen in parallel with any instruction in retrieveCoins

(L18–L27), since retrieveCoins must be finished in the **await** at L38. Similarly, it also reveals that showCoin (L3) cannot happen in parallel with showIncome. On the other hand, SACO detects that the **await** (L24) and the assignment (L16) may happen in parallel. This could be a problem for the termination of retrieveCoins, as the shared variable coins that controls the loop may be modified in parallel, but our termination analysis can overcome this difficulty. Since the result of the MHP analysis refines the control-flow, we could also consider applying the MHP and points-to analyses continuously to refine the results of each other. In SACO we apply them only once.

3 Advanced Analyses

Termination Analysis. The main challenge is in handling *shared-memory* concurrent programs. When execution interleaves from one task to another, the shared-memory may be modified by the interleaved task. The modifications can affect the behavior of the program and change its termination behavior and its resource consumption. Inspired by the rely-guarantee principle used for compositional verification and analysis [12, 5] of thread-based concurrent programs, SACO incorporates a novel termination analysis for concurrent objects [4] which assumes a *property* on the global state in order to prove termination of a loop and, then, proves that this property holds. The property to prove is the *finiteness* of the shared-data involved in the termination proof, i.e., proving that such shared-memory is updated a finite number of times. Our analysis is based on a circular style of reasoning since the finiteness assumptions are proved by proving termination of the loops in which that shared-memory is modified. Crucial for accuracy is the use of the information inferred by the MHP analysis which allows us to restrict the set of program points on which the property has to be proved to those that may actually interleave its execution with the considered loop.

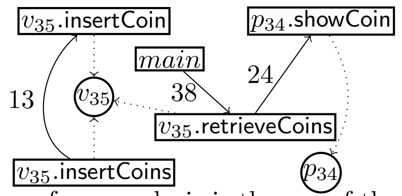
Consider the function retrieveCoins from Sec. 2. At the **await** (L24) the value of the shared variable coins may change, since other tasks may take the object's lock and modify coins. In order to prove termination, the analysis first assumes that coins is updated a finite number of times. Under this assumption the loop is terminating because eventually the value of coins will stop being updated by other tasks, and then it will decrease at each iteration of the loop. The second step is to prove that the assumption holds, i.e., that the instructions updating coins are executed a finite number of times. The only update instruction that may happen in parallel with the **await** is in insertCoin (L16), which is called from insertCoins and this from main. Since these three functions are terminating (their termination can be proved without any assumption), the assumption holds and therefore retrieveCoins terminates. Similarly, the analysis can prove the termination of the other functions, thus proving the whole program terminating.

Resource Analysis. SACO can measure different types of costs (e.g., number of execution steps, memory created, etc.) [2]. In the output, it returns upper bounds on the worst-case cost of executing the concurrent program. The results of our termination analysis provide useful information for cost: if the program is terminating then the size of all data is bounded (we use x^+ to refer to the maximal value for x). Thus, we can give cost bounds in terms of the maximum and/or minimum values that the involved data can reach. Still, we need novel techniques to infer upper bounds on the number of iterations of loops whose execution might interleave with instructions that update the shared memory. SACO incorporates a novel approach which is based on the combination of *local* ranking functions (i.e., ranking functions obtained by ignoring the concurrent interleaving behaviors) with upper bounds on the *number of visits* to the instructions which update the shared memory. As in termination, the function $\mathcal{MHP}$ is used to restrict the set of points whose visits have to be counted to those that indeed may interleave.

Consider again the loop inside retrieveCoins. Ignoring concurrent interleavings, a local ranking function $RF =$ coins is easily computed. In order to obtain an upper bound on the number of iterations considering interleavings, we need to calculate the number of visits to L16, the only instruction that updates coins and MHP with the **await** in L24. We need to add the number of visits to L16 for every path of calls reaching it, in this case main–insertCoins–insertCoin only. By applying the analysis recursively we obtain that L16 is visited n times. Combining the local ranking function and the number of visits to L16 we obtain that an upper bound on the number of iterations of the loop in retrieveCoins is $\text{coin}^+{*}\text{n}$.

Finally, we use the results of points-to analysis to infer the cost at the level of the distributed components (i.e., the objects). Namely, we give an upper bound of the form $c(\epsilon)*(\dots)+c(35)*(\text{coin}^+{*}\text{n}\dots)+c(34)*(\dots)$ which distinguishes the cost attributed to each abstract object o by means of its associated marker $c(o)$.

Deadlock Analysis. The combination of non-blocking (**await**) and blocking (**get**) mechanisms to access futures may give rise to complex deadlock situations. SACO provides a rigorous formal analysis which ensures deadlock freedom, as described in [6]. Similarly to other deadlock analyses, our analysis is based on constructing a *dependency graph* which, if acyclic, guarantees that the program is deadlock free. In order to construct the dependency graph, we use points-to analysis to identify the set of objects and tasks created along any execution. Given this information, the construction of the graph is done by a traversal of the program in which we analyze **await** and **get** instructions in order to detect possible deadlock situations. However, without further *temporal* information, our dependency graphs would be extremely imprecise. The crux of our analysis is the use of the MHP analysis which allows us to label the dependency graph with the program points of the synchronization instructions that introduce the dependencies and, thus, that may potentially induce deadlocks. In a post-process, we discard *unfeasible* cycles in which the synchronization instructions involved in the circular dependency cannot happen in parallel. The dependency graph for our example is shown above. Circular nodes represent objects and squares tasks. Solid edges are



tagged with the program point that generated them (**await** or **get** instructions). Dotted edges go from each task to their objects indicating ownership. In our example, there are no cycles in the graph. Thus, the program is deadlock free.

4 Related Tools and Conclusions

We have presented a powerful static analyzer for an actor-based concurrency model, which is lately regaining much attention due to its adoption in Erlang, Scala and concurrent objects (e.g., there are libraries in Java implementing concurrent objects). As regards to related tools, there is another tool [7] which performs deadlock analysis of concurrent objects but, unlike SACO, it does not rely on MHP and points-to analyses. We refer to [3, 6] for detailed descriptions on the false positives that our tool can give. Regarding termination, we only know of the TERMINATOR tool [8] for thread-based concurrency. As far as we know, there are no other cost analyzers for imperative concurrent programs.

Acknowledgments. This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu) and by the Spanish projects TIN2008-05624 and TIN2012-38137.

References

1. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *APLAS'11*, LNCS 7078, pp. 238-254. Springer, 2011.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *FORTE'12*, LNCS 7273, pp. 35-51. Springer, 2012.
4. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA'13*, LNCS 8172, pp. 349-364. Springer, 2013.
5. B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *PLDI'07, pp. 320–330.* ACM, 2007.
6. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS 7892, pp. 273-288. Springer, 2013.
7. E. Giachino and C. Laneve. Analysis of Deadlocks in Object Groups. In *FMOODS/FORTE*, LNCS 6722, pp. 168-182. Springer, 2011.
8. http://research.microsoft.com/en us/um/cambridge/projects/terminator/.
9. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO'10 (Revised Papers)*, LNCS 6957, pp. 142-164. Springer, 2012.
10. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *PPoPP'10, pp. 25-36.* ACM, 2010.
11. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.
12. C. Popeea and A. Rybalchenko. Compositional Termination Proofs for Multi-Threaded Programs. In *TACAS'12*, LNCS 7214, pp. 237-251. Springer, 2012.