J Autom Reasoning (2017) 59:47–85
DOI 10.1007/s10817-016-9400-6

Rely-Guarantee Termination and Cost Analyses of Loops with Concurrent Interleavings

Elvira Albert[1] · **Antonio Flores-Montoya**[2] · **Samir Genaim**[1] · **Enrique Martin-Martin**[1]

Received: 28 November 2016 / Accepted: 29 November 2016 / Published online: 17 December 2016

Abstract By following a *rely-guarantee* style of reasoning, we present novel termination and cost analyses for concurrent programs that, in order to prove termination or infer the cost of a considered loop: (1) infer the termination/cost of each loop as if it were a sequential one, imposing assertions on how shared-data is modified concurrently; and then (2) prove that these assertions cannot be violated infinitely many times and, for cost analysis, infer how many times they are violated. At the core of the analysis, we use a *may-happen-in-parallel* analysis to restrict the set of program points whose execution can interleave. Interestingly, the same kind of reasoning can be applied to prove termination and infer *upper bounds* on the number of iterations of loops with concurrent interleavings. To the best of our knowledge, this is the first method to automatically bound the cost of such kind of loops. We have implemented our analysis for an *actor-based* language, and showed its accuracy and efficiency by applying it on several typical applications for concurrent programs and on an industrial case study.

Keywords Static analysis · Actor model · Concurrency · Rely-guarantee · Termination analysis · Cost analysis · May-happen-in-parallel analysis

This work was funded partially by the EU Project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO projects TIN2012-38137 and TIN2015-69175-C4-2-R, and by the CM project S2013/ICE-3006.

✉ Enrique Martin-Martin
emartinm@ucm.es

Elvira Albert
elvira@fdi.ucm.es

Antonio Flores-Montoya
aeflores@cs.tu-darmstadt.de

Samir Genaim
samir.genaim@fdi.ucm.es

1 Complutense University of Madrid (UCM), Madrid, Spain

2 Technische Universität Darmstadt (TUD), Darmstadt, Germany

1 Introduction

Proving termination and inferring the cost of concurrent programs is challenging mainly due to loops with *concurrent interleavings*. Those are loops whose executions can interleave, for instance, during the execution of a loop it can suspend and the processor be given to another loop, and in turn the execution of this second loop can suspend and the execution of the first loop resumes, and so on. Such interleavings can happen one or multiple times, and can involve two or more loops. Advanced techniques are needed in order to automatically infer the termination and cost in the presence of interleaved executions and a shared memory which can be modified during the concurrent interleavings. This kind of loops arise in any concurrent language and most of the basic ideas pursued in this article would be adaptable. For the sake of concreteness, we develop our approach for the *concurrent objects* concurrency model and use the core subset of the ABS language [27]. However, our implementation supports the complete ABS language (with the exception of recursive methods).

The *actor*-based paradigm [1] on which concurrent objects are based has evolved as a powerful computational model for defining distributed and concurrent systems. In this paradigm, actors are the universal primitives of concurrent computation: in response to a message, an actor can make local decisions, create more actors, send more messages and determine how to respond to the next message received. *Concurrent objects* (a.k.a. active objects) [30,33] are actors which communicate via *asynchronous* method calls. 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 a task has to release the object lock explicitly . 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 to execute. The synchronization between the caller and the callee methods can be performed when the result is necessary by means of *future variables* [21]—see lines 10 and 11 of the program in Fig. 2 for an example of a future variable-based synchronization. Note that in this concurrency model instructions are atomic, there are no race conditions and no weak memory. The underlying concurrency model of actor languages forms the basis of the programming languages Erlang [16] and Scala [26] that have gained in popularity, in part due to their support for scalable concurrency, and of JavaScript. There are also implementations of actor libraries for Java.

Termination analysis of concurrent and distributed systems is receiving considerable attention [2,4,19,29]. The main challenge is in handling *shared-memory* concurrent programs. This is because, when execution interleaves from one task to another, the shared-memory may be modified by the interleaved task. The modifications will affect the behavior of the program and, in particular, can change its termination behavior and its resource consumption. Inspired by the rely-guarantee style of reasoning, used for compositional verification [22] and analysis [19] of thread-based concurrent programs, we present a novel termination analysis for concurrent objects 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 we propose is that some assertions that we synthesize on the shared-memory, that are essential for proving sequential termination, are violated at most a finite number of times during the interleavings. We show that this is enough to guarantee termination in a concurrent setting. We refer to this property as the *finiteness* property. In contrast to the classical rely-guarantee reasoning, the properties that our approach assumes (the finiteness properties) and proves (the termination of loops) are liveness properties. An example of such assertions is that the value of a field does not increase. The intuition about our method is that

if an assertion is violated a finite number of times, there will be a point from which it will not be violated any longer, and thus a termination proof that uses this assertion is valid. Our method is based on a circular style of reasoning since the finiteness assumptions are proved by proving termination of the loops in which the assertions are violated. Crucial for accuracy is the use of the information inferred by a *may-happen-in-parallel* (MHP) analysis [10], 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.

Besides termination, we also are able to apply this style of reasoning in order to infer the resource consumption (or cost) of executing the concurrent program. The results of our termination analysis already provide useful information for cost: if the program is terminating, we know that the size of all data is bounded. 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. We provide 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 the case of the termination analysis, an auxiliary MHP analysis is used to restrict the set of points whose visits have to be counted to those that indeed may interleave. Both the termination and the cost analyses consider unbounded integers as commonly done in the literature. To the best of our knowledge this is the first approach to infer the cost of loops with concurrent interleavings.

1.1 Summary of Contributions

The main technical contributions of this article are:

- A novel termination analysis which can prove termination by assuming some properties about how the fields are modified (or not) by the environment and then proves that these properties eventually hold. Essentially, our method translates the concurrent program into a sequential setting using the assumptions and then tries to prove the assumptions correct.
- A novel cost analysis for loops with concurrent interleavings which is based on the observation that, provided the instruction(s) that interleave(s) is executed a finite number of times, a bound for the loops can be computed as: the maximum number of iterations of the loop ignoring the interleaved instructions, but assuming that such instructions update the shared memory with its maximum value, multiplied by the maximum number of times that the instructions that interleave can be executed. We also present several refinements to this basic idea in order to improve the precision for some common patterns.
- An implementation of our analyses in the SACO system [3], a Static Analyzer for Concurrent Objects. Experimental evaluation of the termination analysis has been performed on a case study developed by Fredhopper® and several other smaller applications. Preliminary results are promising in both the accuracy and efficiency of the analysis.

A preliminary version of this paper has been published in the proceedings of ATVA'13 [11]. The novelties of this article are: (1) The termination and cost algorithms have been improved to generalized finiteness assumptions, rather than considering only the assumption that fields are not modified. This substantially enlarges the class of programs we can prove terminating and bound their cost. (2) The algorithm that bounds the number of iterations of loops has been refined to obtain smaller bounds for certain types of interleavings with the considered loop.

(3) The cost analysis has been implemented and applied on several programs. (4) Complete and revised proofs have been included in the "Appendix".

1.2 Organization of the Article

The rest of the article is organized as follows. Section 2 contains preliminaries about the language, termination and cost, and the notion of scope which is used by our analysis.

Section 3 explains the rely-guarantee termination analysis. It starts by illustrating the basic reasoning on some simple examples that are challenging for termination. It then presents the main components of our termination analyzer in an algorithmic way. Finally, we provide an automatic method to infer field boundedness, i.e., knowing that the fields have an upper and lower bound.

Section 4 presents our rely-guarantee cost analysis. We start by summarizing the technique of cost analysis in the context of sequential programs. As for termination, the basic reasoning behind our analysis is then explained on some simple programs. We then proceed to present the algorithm to obtain the upper bounds on the number of iterations for loops with concurrent interleavings, and later describe several improvements to the basic reasoning.

Section 5 contains an experimental evaluation of the analyses using the SACO system. Finally, Sect. 6 summarizes related work and Sect. 7 presents the conclusions and future work.

2 Preliminaries

This section presents the syntax and semantics of the language considered in this paper, as well as the notions of termination, cost and scopes that will be used throughout this paper. The language used is basically the *Abstract Behavioral Specification* (ABS) language [4,27], with some unnecessary details omitted for the sake of simplifying the presentation.

2.1 Syntax

A *program* consists of a set of classes, where each defines a set of fields and a set of methods. The set of valid types includes the class names, primitives types such as **int** and **void**, functional types and *future* variable types of the form $fut\langle T\rangle$ where T is a valid type. For simplicity, when writing programs, we omit definitions of functional types and their corresponding constructors and accessors as they are quite standard. In what follows, the notation $\bar{a}$ is used as a shorthand for $a_1, ..., a_n$, where a_i are some syntactic objects. The syntax of a class declaration CL, a method declaration M and an instruction s is as follows:

$$
\begin{array}{lll}
CL & ::= & \textbf{class}\ C\ \{\overline{T\ f};\ \overline{M}\} \\
M & ::= & T\ m(\overline{T\ x})\ \{\overline{T\ y};\ s;\ \} \\
s & ::= & s; s \mid z = e \mid \textbf{this}.f = e \mid \textbf{await}\ z? \mid \textbf{if}\ b\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ b\ \textbf{do}\ s \mid \textbf{return}\ z \\
e & ::= & \textbf{new}\ C(\bar{z}) \mid z!m(\bar{z}) \mid pu \mid \textbf{this} \mid z \mid \textbf{this}.f
\end{array}
$$

where: (1) C, T and m are class, type and method names respectively; (2) $\overline{T\ f}$ is a list of fields defined in class C—all field are private; (3) $\overline{T\ x}$ and $\overline{T\ y}$ are lists of formal parameters and local variables of method m—we often refer to both as local variables; (4) z is a parameter or a local variable; (5) b is a Boolean expression whose details are omitted; (6) **this** is a special variable that refers to the object on which it is evaluated; and (7) pu is a pure expression, e.g., an arithmetic expression, **null**, or an access to a functional data-structure. Functional data-structures in ABS are immutable [27]: updates can be made to variables and fields, but

not to the content of the data-structure to which they point. We assume that all execution paths of a given method end with a **return** instruction. Programs should include a method *main* from which the execution (and the analyses) start. The set of fields of a given class is often referred to as the *shared variables*. For the sake of brevity, when we are not interested in the return value of a method call $x = y!m(\bar{z})$, we write it as $y!m(\bar{z})$.

2.2 Concurrency Model and Semantics

As in the actor-model, the main idea is that control and data are encapsulated within the notion of concurrent object. Thus each object encapsulates a *local heap* which stores the data that is *shared* within the object (i.e., values of fields). Fields are always accessed using the **this** object, and any other object can only access such fields through method calls. The concurrency model is as follows. Each object has a lock that is shared by all tasks that belong to the object. A task is created on some object when one of its methods is called, thus, a task is in principle an instance of the corresponding method plus a local state that assigns values to its local variables. Synchronization between tasks is done by means of future variables. An **await** z? instruction is used to synchronize with the task created due to the invocation $z{=}x!m(\bar{z})$. In such case, **await** z? blocks if the future variable z is not available, i.e., the corresponding task is not finished. In the meantime, the object's lock can be released and some other *pending* task on that object can take it. Program points that contain **await** instructions are referred to as *release points*.

A *program state* St is a set $St = \mathtt{Ob} \cup \mathtt{T}$ where $\mathtt{Ob}$ is the set of all created objects, and $\mathtt{T}$ is the set of all created tasks. An *object* is a term $ob(o, a, lk)$ where o is a unique object identifier, a is a mapping from the object fields to their values and lk the identifier of the *active task* that holds the object's lock or $\perp$ if the object's lock is free. Only one task can be *active* (running) in each object and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they have terminated and released the lock. A *task* is a term $tsk(t, m, o, l, s)$ where t is a unique task identifier, m is the method name executing in the task, o identifies the object to which the task belongs, l is a mapping from local (possibly future) variables to their values and s is the sequence of instructions to be executed (enclosed by $\langle$ and $\rangle$) or $s \equiv \epsilon(v)$ if the task has terminated and the return value v is available. Created objects and tasks never disappear from the state.

The execution of a program starts from the initial state that takes the form $St_0 = \{ob(0, a, 0), tsk(0, \mathsf{main}, 0, l, \langle body(\mathsf{main})\rangle)\}$, which includes an initial object with identifier 0 executing task 0. The mapping a is empty since main has no fields, l maps the parameters of main to their initial values and other local variables to **null** or 0 depending on their type and $body(\mathsf{main})$ refers to the sequence of instructions in the method main preceded by an auxiliary instruction take. This auxiliary instruction will allow the task to fetch the lock. The execution proceeds from St_0 by applying *non-deterministically* the semantic rules depicted in Fig. 1 (the execution of sequential instructions is standard and thus omitted). The operational semantics is given in a rewriting-based style where a step is a transition of the form

$$X \underset{\cdots}{Y} \rightarrow \underset{\cdots}{Y'} Z$$

in which: dotted underlining indicates that term Y is rewritten to Y'; we look up the term X but do not modify it and hence it is not included in the right-hand side; and term Z is newly added to the state. Transitions are applied according to the rules:

– (NEW): An active task t in object o creates an object o' of type B. The object's fields are initialized to default values, and, assuming k is the length of $\bar{z}$, its first k fields are initialized to the value of the parameters $\bar{z}$. Finally, o' is introduced to the state with a free

$$\frac{\text{fresh}(o'),\ l' = l[x \to o'],\ a' = \text{init\_atts}(B, l(\overline{z}))}{ob(o,a,t)\ tsk(t,m,o,l,\langle x = \textbf{new}\ B(\overline{z}); s\rangle) \to tsk(t,m,o,l',\langle s\rangle)\ ob(o',a',\bot)}\ (\textsc{New})$$

$$\frac{}{ob(o,a,\bot)\ tsk(t,m,o,l,\langle \mathsf{take}; s\rangle) \to ob(o,a,t)\ tsk(t,m,o,l,\langle s\rangle)}\ (\textsc{Activate})$$

$$\frac{l(x) = o_1,\ o_1 \neq \textbf{null},\ \text{fresh}(t_1),\ l' = l[y \to t_1],\ l_1 = buildLocals(l(\overline{z}), m_1)}{ob(o,a,t)\ tsk(t,m,o,l,\langle y = x!m_1(\overline{z}); s\rangle) \to tsk(t,m,o,l',\langle s\rangle)\ tsk(t_1,m_1,o_1,l_1,\langle body(m_1)\rangle)}\ (\textsc{Async-Call})$$

$$\frac{l(y) = t_1, (lk = t \vee lk = \bot)}{ob(o,a,lk)\ tsk(t,m,o,l,\langle \textbf{await}\ y?; s\rangle)\ tsk(t_1,m_1,o_1,l_1,\epsilon(v)) \to ob(o,a,t)\ tsk(t,m,o,l,\langle s\rangle)}\ (\textsc{Await1})$$

$$\frac{l(y) = t_1, \langle s_1\rangle \neq \epsilon(v)}{ob(o,a,t)\ tsk(t,m,o,l,\langle \textbf{await}\ y?; s\rangle)\ tsk(t_1,m_1,o_1,l_1,\langle s_1\rangle) \to ob(o,a,\bot)\ tsk(t,m,o,l,\langle \textbf{await}\ y?; s\rangle)}\ (\textsc{Await2})$$

$$\frac{v = l(x)}{ob(o,a,t)\ tsk(t,m,o,l,\langle \textbf{return}\ x\rangle) \to ob(o,a,\bot)\ tsk(t,m,o,l,\epsilon(v))}\ (\textsc{Return})$$

Fig. 1 Summarized semantics of concurrent objects

lock, and variable x is assigned a reference to o' in the local variables map l'. Observe that as the previous object o is not modified, it is not included in the resulting state.

- (ACTIVATE): A newly created task can obtain the corresponding object's lock if it is free by consuming the auxiliary instruction take.
- (ASYNC- CALL): A method call creates a new task (the initial state is created by *build-Locals*) with a fresh task identifier t_1 which is associated to the corresponding future variable y in l'. Note that $body(m_1)$ returns the sequence of instructions of m_1 preceded by a special symbol take.
- (AWAIT1): If the future variable we are awaiting for points to a finished task, the **await** can be completed. This rule can be applied if the task t has the object's lock or if the object's lock is free. In the latter case, the lock is taken by the task t. The finished task t_1 is only looked up but it does not disappear from the state as its return value may be needed later on. The return value v can be accessed using the future variable y at any time after the execution of the **await** y?[1]
- (AWAIT2): If the future variable points to an unfinished task, i.e., a task whose sequence of instructions $\langle s_1\rangle$ is different from $\epsilon(v)$, then the task yields the lock so any other task of the same object can take it.
- (RETURN): When **return** is executed, the return value v is left in the last argument of the tsk object so that it can be obtained through the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished*—marked by adding $\epsilon(v)$—but it does not disappear from the state.

Example 1 Figure 2 shows some simple examples which will illustrate different aspects of our analysis. We have two classes Task and Server whose definitions have not been included. They represent tasks that have to be performed and servers that can perform those tasks. We assume that class Server has a method with the signature "**void** startTask(Task tk)" that

[1] For simplicity of the presentation, we have not included (nor used in the examples) an additional instruction to get the return value of an asynchronous call. This instruction exists in the ABS language [27] and it is called y.get, where y is the future variable, but it does not introduce any challenge to the analysis.

```
1  class TaskQueue{
2    List<Task> pending;
3
4    void addTasks(List<Task> list) {
5        Fut<void> f;
6        Task t;
7        while (list != Nil) {
8            t = head(list);
9            list = tail(list);
10           f = this!addTask(t);
11           await f?;
12       }
13   }
14
15   void addTask(Task t) {
16       pending = appendright(pending,t);
17   }
18
19   void consumeAsync(List<Server> srvs){
20       Task t;
21       Server p;
22       List<Server> srvs_it;
23       while (pending != Nil) {
24           t = head(pending);
25           pending = tail(pending);
26           srvs_it = srvs;
27           while(srvs_it != Nil) {
28               p = head(srvs_it);
29               p!startTask(t);
30               srvs_it = tail(srvs_it);
31           }
32       }
33   }
34
35
36   void consumeSync(List<Server> srvs) {
37       Task t;
38       Server p;
39       Fut<void> f;
40       while (pending != Nil) {
41           t = head(pending);
42           pending = tail(pending);
43           p = head(srvs);
44           f = p!startTask(t);
45           await f?;
46       }
47   }
48 } // end of class TaskQueue
49
50 // Main Methods
51 main1(List<Task> tsks, List<Server> srvs) {
52   TaskQueue q;
53   q = new TaskQueue();
54   q!addTasks(tsks);
55   q!consumeAsync(srvs);
56 }
57
58 main2(List<Task> tsks, List<Server> srvs) {
59   TaskQueue q;
60   Fut<void> f;
61   q = new TaskQueue();
62   f = q!addTasks(tsks);
63   await f?;
64   q!consumeSync(srvs);
65 }
66
67 main3(List<Task> tsks, List<Server> srvs) {
68   TaskQueue q;
69   q = new TaskQueue();
70   q!addTasks(tsks);
71   q!consumeSync(srvs);
72 }
```

Fig. 2 Simple examples for termination and cost

receives a task and executes it. Additionally, we have class TaskQueue which implements a queue of tasks to which one can add a single task using method addTask or a list of tasks using method addTasks. The loop that adds the tasks invokes asynchronously method addTask and then awaits for its termination at Line 11 (L11 for short). We use the predefined generic type List<E> with the usual operations appendright to add an element of type E to the end of the list, head to get the element in the head of the list, tail to get the remaining elements and the constructor Nil for empty lists. These operations are performed on pure data (i.e., data that possibly contains references but does not access the shared memory) and are executed sequentially. The class has two other methods, consumeAsync and consumeSync, to consume the tasks inside the queue with a given list of servers srvs. The former method starts all tasks (L29) concurrently in all the servers. Instead, method consumeSync executes each task synchronously only in the first server of the list ps. It releases the processor and waits until the task is finished at L45. In the rightmost column, there are 3 implementations of main methods which are defined outside the classes. In Fig. 3 we show some execution steps from main3, where we assume that main and other **void** methods finish with a **return void** instruction and every local mapping l contains a built-in symbol **void** defined as [**void** $\mapsto \bot$]. We have

$$
\begin{aligned}
St_0 \equiv\ & \{obj(0,a_0,0),\ tsk(0,\mathsf{main3},0,l_0,\langle \underline{\mathsf{q = new\ TaskQueue();}}\ \mathsf{q!addTasks(tsks);}\\
& \mathsf{q!consumeSync(srvs);\ return\ void;}\rangle)\} \xrightarrow{\textsc{New}}\\
St_1 \equiv\ & \{obj(0,a_0,0),\ obj(1,a_1,\bot),\ tsk(0,\mathsf{main3},0,l_0',\langle \underline{\mathsf{q!addTasks(tsks);}}\\
& \mathsf{q!consumeSync(srvs);\ return\ void;}\rangle)\} \xrightarrow{\textsc{Async-Call}}\\
St_2 \equiv\ & \{obj(0,a_0,0),\ obj(1,a_1,\bot),\ tsk(0,\mathsf{main3},0,l_0',\langle \underline{\mathsf{q!consumeSync(srvs);}}\\
& \mathsf{return\ void;}\rangle),\ tsk(1,\mathsf{addTasks},1,l_1,\langle \mathsf{take;\ while\ (list\ !=\ Nil)\{\ldots\}}\rangle)\\
& \xrightarrow{\textsc{Async-Call}}\\
St_3 \equiv\ & \{obj(0,a_0,0),\ obj(1,a_1,\bot),\ tsk(0,\mathsf{main3},0,l_0',\langle \underline{\mathsf{return\ void;}}\rangle),\\
& tsk(1,\mathsf{addTasks},1,l_1,\langle \mathsf{take;\ while\ (list\ !=\ Nil)\{\ldots\}}\rangle),\\
& tsk(2,\mathsf{consumeSync},1,l_2,\langle \mathsf{take;\ while\ (pending\ !=\ Nil)\{\ldots\}}\rangle)\\
& \xrightarrow{\textsc{Return}}\\
St_4 \equiv\ & \{obj(0,a_0,\bot),\ obj(1,a_1,\bot),\ tsk(0,\mathsf{main3},0,l_0',\epsilon(\bot)),\\
& tsk(1,\mathsf{addTasks},1,l_1,\langle \underline{\mathsf{take;}}\ \mathsf{while\ (list\ !=\ Nil)\{\ldots\}}\rangle),\\
& tsk(2,\mathsf{consumeSync},1,l_2,\langle \mathsf{take;\ while\ (pending\ !=\ Nil)\{\ldots\}}\rangle)\\
& \xrightarrow{\textsc{Activate}}\\
St_5 \equiv\ & \{obj(0,a_0,\bot),\ obj(1,a_1,1),\ tsk(0,\mathsf{main3},0,l_0',\epsilon(\bot)),\\
& tsk(1,\mathsf{addTasks},1,l_1,\langle \mathsf{while\ (list\ !=\ Nil)\{\ldots\}}\rangle),\\
& tsk(2,\mathsf{consumeSync},1,l_2,\langle \mathsf{take;\ while\ (pending\ !=\ Nil)\{\ldots\}}\rangle)
\end{aligned}
$$

Fig. 3 Executions steps starting from main3

underlined the instruction that is consumed in every step. Observe that the execution of **new** at St_0 creates the object identified by 1. Then, the executions of the asynchronous calls at St_1 and St_2 spawn new tasks on object 1 identified by 1 and 2, respectively. In St_3, we perform a (RETURN) step that stores the value $\bot$ of the built-in variable **void** as the result of task 0, which terminates and object 0 becomes idle. Then, in St_4, object 1 (which was idle) selects task 1 for execution. Note that as scheduling is non-deterministic, any of both pending tasks (1 or 2) could have been selected.

2.3 Termination and Cost

Traces take the form $tr \equiv St_0 \rightarrow^{b_0} \cdots \rightarrow^{b_{n-1}} St_n$ where (1) St_0 is an initial state, i.e., it includes a single task corresponding to the main method; (2) each transition $St_i \rightarrow^{b_i} St_{i+1}$ corresponds to a semantic rule application; and (3) the superscript b_i is the instruction executed to move from state St_i to state St_{i+1}. Note that all semantic rules complete an instruction except AWAIT2 which has no superscript. A trace is *complete* if no task in St_n can progress. A trace is *finished* if every task $tsk(t,m,o,l,s) \in St_n$ is finished, i.e., $s = \epsilon(v)$. If a trace is complete but not finished, then its last state must be *deadlocked* (or erroneous, e.g., trying to dereference **null**). Deadlocks happen when several tasks are awaiting for each other to terminate and remain blocked. Deadlock is different from non-termination, as non-terminating traces keep on consuming instructions. Note that from a given state there may be several possible *non-deterministic* execution steps that can be taken (since we have no assumptions on scheduling). We say that a program is *terminating* if there are no infinite traces starting from St_0, i.e., every non-complete trace from St_0 will eventually become complete.

When measuring the cost of a trace, different metrics can be considered. Such metrics are typically called *cost models*. A cost model is a function $\mathcal{M} : Ins \mapsto \mathbb{R}^+$ which maps instructions to positive real numbers (*Ins* is the set of valid instructions as in the grammar

above). Note that we exclude from our study cost models that take into account the external environment (e.g., cache misses, or memory contention) and focus exclusively on cost models that are observable from the program itself. The cost of an execution step is defined as $\mathcal{M}(St \rightarrow^b St') = \mathcal{M}(b)$, i.e., the cost of the instruction used to perform the corresponding step. The cost of execution steps corresponding to AWAIT2 and ACTIVATE is 0. The cost of a trace is the sum of the costs of all its execution steps. The cost of executing a program is the *maximum* of the costs of all possible traces. We aim at inferring an *upper bound* on the cost of executing a program P for the defined cost model, denoted $\mathtt{UB}_P$, which is at least as that maximum.

Example 2 A cost model that counts the number of instructions is defined as $\mathcal{M}_{inst}(b) = 1$ where b is any instruction. A cost model that counts the number of visits to a method m is defined as $\mathcal{M}_{visits\_m}(b) = 1$ if $b \equiv x!m(\bar{z})$ and 0 otherwise. Consider the partial trace of Example 1. By applying $\mathcal{M}_{inst}$ we get 4 executed instructions (as the application of ACTIVATE does not involve any instruction) and if we count $\mathcal{M}_{visits\_ConsumSync}$ we obtain 1.

2.4 Programs and Scopes

The algorithms developed in this article operate on a set of scopes, denoted $\mathcal{SS}et$, that we extract (syntactically) from the input program. We distinguish two kinds of scopes: methods and loops. A *loop scope* has the form $S_l \equiv$ **while** b **do** s and a *method scope* the form $S_m \equiv T\ m(\overline{T\ x})\{\overline{T\ y}; s;\}$. Note that scopes can be nested. We say that an instruction s *belongs* to a scope S, denoted by $s \in S$, if S is the smallest scope in which s appears syntactically. An instruction can appear inside several scopes but it can only belong to one. A loop scope $S_l \equiv$ **while** p **do** s is contained in a scope S (denoted $S \rightarrow S_l$) if (**while** p **do** s) $\in S$. A method scope S_m is contained in a scope S (denoted $S \xrightarrow{\text{LN}} S_m$) if there is a method call instruction $z!m(\bar{z}) \in S$ at line LN.

We assume there are no recursive methods. In that case, the set of scopes $\mathcal{SS}et$ and the containment relationship $\rightarrow$ form a directed acyclic graph (DAG) in which the main method's scope S_{main} is the outermost one (it does not belong to any scope). Note that the DAG can have multiple edges from one scope S to another S' if there are several method calls to S' that belong to S. Given a scope S, *Subprogram*(S) is the program defined by S and all the scopes transitively contained in S.

Example 3 Consider the program depicted in Fig. 2. If we consider main1 as the main method, then the set of scopes is

$$\mathcal{SS}et = \{S_{\mathtt{main1}}, S_{\mathtt{addTasks}}, S_7, S_{\mathtt{addTask}}, S_{\mathtt{consumeAsync}}, S_{23}, S_{27}, S_{\mathtt{startTask}}\}$$

where S_7, S_{23} and S_{27} are the scopes of the loops defined at L7, L23 and L27. The relationships among scopes are represented in the following graph:

$$S_{\mathtt{main1}} \xrightarrow{54} S_{\mathtt{addTasks}} \longrightarrow S_7 \xrightarrow{10} S_{\mathtt{addTask}}$$
$$S_{\mathtt{main1}} \xrightarrow{55} S_{\mathtt{consumeAsync}} \rightarrow S_{23} \longrightarrow S_{27} \xrightarrow{29} S_{\mathtt{startTask}}$$

The instruction at L28 appears in the scopes $S_{\mathtt{consumeAsync}}$, S_{23} and S_{27} but it only *belongs* to S_{27}. The subprogram of S_{23} is *Subprogram*$(S_{23}) = \{S_{23}, S_{27}, S_{\mathtt{startTask}}\}$.

Next we define the notion of *local termination of scopes*. With this notion we aim at inspecting the termination of each scope independently from the termination of its inner or outer scopes. This notion is also useful for bounding the number of iterations of a loop

independently from how many times this loop is reached. These bounds can then be composed to infer the overall cost of the program. Let us define this notion formally.

Assume that each loop scope S_l executed in a task t is instrumented with a local variable $i_{l,t}$ that acts as a loop counter such that (1) it is set to 0 before entering the loop; and (2) it is increased by 1 at the beginning of each iteration. Moreover, each method scope S_m executed in a task t is instrumented with a counter $i_{m,t}$ that is set to 1 at the entry of method m. Given a trace tr and a scope S_x executed in a task t, we let $sup(tr, i_{x,t})$ be the supremum of the values that $i_{x,t}$ can take in tr.

Definition 1 *(Locally terminating scope)* We say that a scope S_x is *locally terminating* if for any task t in any trace tr there exists $n_{t,tr} \in \mathbb{N}$ such that $sup(tr, i_{x,t}) \leq n_{t,tr}$.

Notice that for any method m, since $i_{m,t}$ is set to 1 at the entry of m and its value does not change, $sup(tr, i_{m,t}) = 1$ for any trace tr and task t; therefore any method is locally terminating. However, these constant counters $i_{m,t}$ allow us to express the definition of local termination uniformly for any scope.

Lemma 1 *If all scopes of a program P are locally terminating, then P is terminating.*

The proof of the above lemma is straightforward. Consequently, our termination analysis (Sect. 3) will attempt to prove termination of each scope of the program. Thanks to Definition 1, when proving termination of a scope, we will be able to substitute its inner scopes by summaries.

In order to compute the cost of a program P, we follow a similar approach, where we first infer local bounds for each scope and then combine them into a function that bounds the cost of P. Suppose that for each scope S we infer a function $\text{LBOUND}_S(\bar{z})$ that bounds $sup(tr, i_{x,t})$ for any tr and any task t (here $\bar{z}$ represents the parameters of method main). These bounds can be composed to compute the cost of P as follows. Let D be the scopes DAG (as in Example 3), an upper bound on the cost of P, with respect to a cost model $\mathcal{M}$, is defined as $\text{UB}_P = \text{UB}(S_{main})$ where:

$$\text{UB}(S_x) = \text{LBOUND}_{S_x} * \left(\sum_{b \in S_x} \mathcal{M}(b) + \sum_{S_x \to S_y \in D} \text{UB}(S_y) \right)$$

Consequently, our cost analysis (Sect. 4) will focus only on inferring LBOUND_S for each scope S. Note that for any method m, the bound $\text{LBOUND}_{S_m} = 1$ is tight since our language is recursion-free. Thus, in the rest of this article such bounds are often refereed to as *loop bounds* since, in the case of loop scopes, they bound the number of iterations of corresponding loops.

3 Termination Analysis

In this section we present our algorithm for proving termination of concurrent programs. We first give the intuition behind our method in Sect. 3.1, and then present our algorithm in Sect. 3.2. In Sect. 3.3 we conclude with some observations on which we rely when developing our cost analysis in Sect. 4.

3.1 Basic Reasoning

Our starting point is an analysis [4] that infers the termination (and resource consumption) of concurrent programs. This analysis works in two phases: (1) in the first one each release point is replaced by instructions of the form $f = *$, for each field f, to indicate that the values of all fields are lost when passing through a release point; and (2) in the second one it tries to prove termination of the different loops as if they were sequential. This approach is clearly sound since it loses any information on the shared-memory when there is a risk of a concurrent interleaving. As a refinement, the tool of [4] allows stating that some fields keep their values at some release points (i.e., $f = *$ is not added for such fields). However, this information is not verified to be correct, and thus the result of the analysis is sound under the condition that such information is sound.

Let us explain the kind of programs that [4] can and cannot prove terminating using the program of Fig. 2, when starting from methods main1, main2 and main3:

- main1: it creates an instance of `TaskQueue`, adds the list of tasks received as input parameter to it and executes consumeAsync to process these tasks. Note that it is not guaranteed that the tasks are added to the queue when consumeAsync starts to execute because, as the call at L54 is not synchronized, the processor can be released at L11 and the method called at L55 can start to execute. Proving termination of the loop in addTasks is straightforward since it does not depend on the shared variable pending. Proving termination of the nested loop of consumeAsync is also straightforward, since it is executed without releasing the processor. Thus, the method of [4] succeeds to prove termination of main1.
- main2: in this case the addition of tasks (i.e., the call to addTasks at L62) is guaranteed to have terminated when consumeSync starts to execute—due to the use of **await** at L63. However, the difficulty is that the loop of consumeSync contains a release point, and thus, if we lose the value of pending at that point, the analysis of [4] would fail to prove termination (since termination of the loop in consumeSync depends on the field pending). The key is to detect that there are no concurrent interleavings at L45 by means of an auxiliary MHP analysis [10], and thus stating that pending keeps its value at that release point.
- main3: since `consumeSync` is called without waiting for completion of `addTasks`, and their corresponding loops have release points, there can be concurrent interleavings. Moreover, since the size of the list pending might increase at the release point of `consumeSync`, the analysis of [4] is not able to prove termination since it is not sound to assume that pending keeps its value. Proving termination of such programs requires developing novel techniques.

In the rest of this section we develop a termination analysis that is able to overcome the problems described above.

Our analysis is also based on the idea of using assertions to describe how fields change their values at release points, however, as will become clear later, in a fundamentally different way from that of [4]. Let us first define the syntax and semantics of such assertions, that we refer to as *shared-memory-assertions* (or simply *assertions* or *assumptions* for brevity).

Definition 2 *(Shared-memory-assertions)* Given a scope S, a shared-memory-assertion C for the release points of S is a set that includes a constraint $f' \bowtie f$, where $\bowtie \in \{=, \geq, \leq, ?\}$, for each field $f \in \mathit{fields}(S)$.

Intuitively, a shared-memory-assertion states how the value of each field might change at the release points. The constraint variables f and f' refer to the values of field f before releasing the processor and after obtaining it back, respectively, and the meaning of the different constraints is as follows: (1) $f' = f$ means that f does not change its value; (2) $f' \geq f$ means that f does not decrease; (3) $f' \leq f$ means that f does not increase; and (4) $f'?f$ means that f can take any value. Note that when a field f is not of type **int**, then f and f' refer to the size of the corresponding data-structure with respect to some predefined size measure (e.g., length of a list or depth of a data-structure).

Given a scope S and an assertion C, in what follows, we assume to have a black-box procedure `seq_termin`(S, C) to prove the local (sequential) termination of S with respect to C as follows: (1) the assertion C is added to the release points of S; (2) each method call $x = y!q(\bar{z})$ that appears in S is replaced by $x = *$, to indicate that the return value can be anything—or we can use a method summary if available; (3) each inner scope of S is replaced by a corresponding loop summary; and (4) any off-the-shelf sequential termination analyzer is used to prove *universal termination* of the resulting code (ignoring release points and using the assertions instead). It is important to note that we prove *universal termination*, which means termination for any initial values for the variables—this is fundamental for the correctness of our approach.

In our termination algorithm, procedure `seq_termin` will be used to prove the local termination of the different scopes. If for some scope S and a corresponding assertion C the call `seq_termin`(S, C) returns *true*, then, under the condition that C is never violated during the execution (of the whole program), we can clearly conclude that S is locally terminating (as in Definition 1). However, we can weaken this condition, and substantially enlarge the class of concurrent programs that we prove terminating.

Observation 1 *(Finiteness assumption) If* `seq_termin`(S, C) *returns true, and in any trace the assertion C is violated only a finite number of times, then S is locally terminating.*

The intuition behind our observation is as follow s. If C is violated only a finite number of times, then there will be a point from which on C is not violated, and thus, since `seq_termin` proves termination for any initial state, we can conclude that S is locally terminating.

Let us see how the above observation can be applied to method main3 of Fig. 2. Recall that the execution of the loops of `consumeSync` and `addTasks` might interleave. The loop of `addTasks` is clearly terminating since it does not depend on the field pending. The loop of `consumeSync` can be shown terminating if we use the assertion $C \equiv \{pending' \leq pending\}$ at its release point. Now since the field pending can be modified only in the loop of `addTasks`, which is terminating, we conclude that C is violated a finite number of times. Thus, according to Observation 1, both loops terminate even if they interleave. Let us conclude with some more examples that explain the basics of our algorithm, before presenting it in the next section.

Example 4 Consider the program of Fig. 4, and note that f is the only shared variable. We assume that method g does not modify f nor has loops and thus we can ignore it. When starting from method main, methods m1, m2 and m3 might execute in parallel, and clearly scopes S_{89}, S_{101} and S_{113} might interleave since each includes a release point. It is easy to see that (1) S_{89} terminates when using the assertion $C_1 \equiv \{f' \leq f\}$, i.e., if field f does not increase at the release point L91, and, moreover, $RF_1(n, m, f) = f$ is a ranking function that witnesses its termination; and (2) S_{101} terminates without any assumption, i.e., with $C_2 \equiv \{f'?f\}$, and $RF_2(n, m, f) = m$ is a ranking function that witnesses its termination; and (3) S_{113} terminates without any assumption, i.e., with $C_3 \equiv \{f'?f\}$, and

```
73 main(int x,int y) {
74   A q;
75   q=new A();
76   q!m1();
77   q!m2(x);
78   q!m3(y);
79 }
80
81 class A {
82   int f;
83
84   void g() {...}
```

```
85   void m1() {
86     Fut<void> x;
87
88     // scope S89
89     while (f>0) {
90       x=this!g();
91       await x?;
92       f=f−1;
93     }
94   }
95
96
```

```
97   void m2(int m) {
98     Fut<void> x;
99
100    // scope S101
101    while (m>0) {
102      x=this!g();
103      await x?;
104      f=∗;
105      m=m−1;
106    }
107  }
108
```

```
109  void m3(int n) {
110    Fut<void> x;
111
112    // scope S113
113    while (n>0) {
114      x=this!g();
115      await x?;
116      f=f−1;
117      n=n−1;
118    }
119  }
120 } \\ end of class A
```

Fig. 4 Examples of interleavings not affecting termination

$RF_3(n, m, f) = n$ is a ranking function that witnesses its termination. Since S_{101} and S_{113} always terminate, and S_{89} cannot interleave with itself (there is only one invocation of m1), we know that f can be modified only a finite number of times at the release point of S_{89}, thus, according to Observation 1, S_{89} terminates when running in parallel with S_{101} since C_1 will be violated only a finite number of times. Note that the lexicographical ranking function $RF_3(n, m, f) = \langle m + n, f \rangle$ witnesses the termination of S_{89}. Now assume that we remove the instruction at L117, which in turn makes S_{113} non-terminating. In this case the update at L116 execute s an infinite number of times, and apparently, we cannot use Observation 1 to conclude that S_{89} is terminating. However, note that the update at L116 does not violate the assertion $C_1 \equiv \{f' \leq f\}$, and thus the termination of S_{113} is not needed to conclude that S_{89} is terminating. Note that, similarly, we can prove termination even when two instances of m1 are running in parallel. This is because the update f=f−1 in m1 does not violate the assertion $C_1 \equiv \{f' \leq f\}$.

3.2 Termination Algorithm

In this section we first present our algorithm for proving local termination of a scope S, and then use it to prove termination of the whole program as in Lemma 1, i.e., by proving that all scopes are locally terminating.

Let us first explain the intuition behind the local termination algorithm. In order to prove that a scope S is locally terminating, we first synthesize an assertion C for its release points (we later explain how this synthesis is done by our algorithm) and call `seq_termin`(S, C). If `seq_termin` returns *true*, then, according to Observation 1, all we need to do is to prove that C can be violated only a finite number of times in any execution. One way to do so is as follows: (1) identify the set of all program points I that modify the shared memory in a way that might violate C, and, moreover, require that such program points can actually happen in parallel with the release points of S; and (2) prove that each program point in I can be reached only a finite number of times, which can be done by recursively calling the local termination algorithm on those scopes that can reach any program point in I. As we will see later, one needs to take care of circular dependencies between scopes as well.

Our algorithm for proving local termination is defined in Algorithm 1, by means of function TERMINATES. The first parameter S is the scope that we want to prove locally terminating, and the second one *SSet* includes the scopes whose local termination requires the local termination of S. The role of the second parameter is to detect circular dependencies, it gets its value in the recursive calls. In order to prove local termination of a scope S, we use the call TERMINATES$(S, \emptyset)$. Let us explain the different lines of the algorithm:

Algorithm 1 MHP-based Termination Analysis

```
1: function TERMINATES(S,SSet)
2:   if S ∈ SSet then return false
3:   LC = field_constraints(S)
4:   for each C ∈ LC do
5:     if seq_termin(S, C) then
6:       RP = release_points(S)
7:       MP = MHP_pairs(RP)
8:       I = field_updates_constr(MP, C)
9:       DepSet = extract_scopes(I)
10:      all_term = true
11:      for each S′ ∈ DepSet do
12:        all_term = all_term ∧ TERMINATES(S′, SSet ∪ {S})
13:      if all_term then
14:        return true
```

1. At L2, if S is in the set *SSet*, then a circular dependency has been detected, i.e., the local termination of S depends on the local termination of S itself. In such a case the algorithm returns *false* (since we cannot handle such cases).
2. At L3, it generates a sequence of candidate assertions $[C_1, C_2, \ldots, C_n]$ for the scope that will be tried in order. Notice that, since the set of fields used in the scope S is finite, it is always possible to generate all the possible shared-memory-assertions. Some heuristics can be used to reduce the list of candidates, though, as we explain later.
3. In the loop starting at L4, the algorithm tries to prove the local termination of S with respect to the different assertions. At L5, it calls `seq_termin`(S, C), and, if successful, it tries to prove that C can be violated only a finite number of times (L6–14).
4. Next it identifies the instructions that might violate C, while S is waiting at a release point, as follows: at L6 it constructs the set *RP* of all release points in S; at L7 it constructs the set *MP* of all program points that may run in parallel with program points in *RP* (this is provided by an auxiliary MHP analysis [10]); and at L8 it remains with $I \subseteq MP$ that actually update a field in $fields(S)$ in a way that might violate C. The set I is a conservative approximation. We perform a simple syntactic analysis to discard those updates that *are guaranteed not to violate* C.
5. Next it proves that program points from I can be reached only a finite number of times as follows: at L9 it constructs a set *DepSet* of *all* scopes that can reach a program point in I, i.e., any scope S' such that a program point of I appears in $Subprogram(S')$. Proving local termination of these scopes guarantees that each instruction in I is executed finitely, and thus C can be violated only a finite number of times.
6. The loop at L11 tries to prove that each scope in *DepSet* is locally terminating. If it finds one that might not locally terminate, it returns *false*. Note that in the recursive call S is added to the second parameter in order to detect circular dependencies.
7. If the algorithm reaches L14, then S is locally terminating, and thus the algorithm returns *true*.

With respect to the generation of candidate assertions, some heuristics can be used in function `field_constraints` to reduce the number and complexity of the generated assertions. A simple improvement is invoking `seq_termin`(S, C_{eq}) with the strongest assertion $C_{eq} = \{f' = f | f \in fields(S)\}$ and obtain its termination proof (e.g., the ranking function RF_{eq}). Instead of generating all the possible assertions with fields in $fields(S)$, it only generates assertions involving those fields that appear in RF_{eq}, since only those fields affect the termination. A better approach—the one implemented in our analysis—inspects the ranking function

RF_{eq} and generates only one assertion that guarantees that the value of the ranking function decreases. For example, considering the ranking function $RF_{eq}(n, f, g) = f - g$ (where f and g are fields and n is a local variable), the generated assertion would be $\{f' \leq f, g' \geq g\}$. If RF_{eq} cannot be computed, then none of these heuristics can be applied, and a sequence $[C_1, \ldots, C_n]$ containing all the possible assertions must be generated. In this case, it is interesting that assertions range from the weakest $C_1 = \{f'?f | f \in fields(S)\}$ to the strongest $C_n = \{f' = f | f \in fields(S)\}$.

Next we apply our termination algorithm to the program of Fig. 2.

Example 5 Consider the program of Fig. 2, and let us show how we prove termination of main2. The challenge is to prove the local termination of `consumeSync`, in particular the loop that starts at L40, let us call the corresponding scope S_{40}. The local termination of all other scopes is straightforward, and thus we skip the corresponding details. Since the condition of the loop in S_{40} depends on the field pending, and, moreover, pending is decreasing in each iteration, at L3 of Algorithm 1 we generate $LC\{\{pending' \leq pending\}\}$ which consists of a single assertion stating that pending does not increase. At L5 of Algorithm 1, the call `seq_termin`$(S_{40}, \{pending' \leq pending\})$ returns *true*, and thus we proceed to prove that $\{pending' \leq pending\}$ can be violated only a finite number of times at the release points $RP = \{\text{L45}\}$. The MHP analysis infers that the update of field pending at L16 (which is the only update) cannot run in parallel with the release point L45. This is due to the use of **await** at lines 11 and 63. Therefore, the set I at L8 of Algorithm 1 is empty and TERMINATES returns *true*.

Example 6 Consider the program of Fig. 2, and let us show how we prove termination of main3. It is similar to what we have done for main2 in Example 5, however, when proving the local termination of scope S_{40}, the MHP analysis infers that the update of field pending at L16 might run in parallel with the release point L45. Moreover, this update might violate $C \equiv \{pending' \leq pending\}$ since it adds new elements to the list pending. To prove that C can be violated only a finite number of times, we need to prove that any scope that can reach L45 is locally terminating. The set of these scopes $DepSet = \{S_{\texttt{addTask}}, S_7, S_{\texttt{addTasks}}, S_{\texttt{main3}}\}$ is generated at L9 of Algorithm 1, and TERMINATES is recursively called for each one. Proving the local termination of S_7 is straightforward since it depends only on the local variable list. The local termination of $S_{\texttt{addTask}}$, $S_{\texttt{addTasks}}$ and $S_{\texttt{main3}}$ is also straightforward since they are (non-recursive) method scopes. We conclude that $S_{\texttt{consumeSync}}$ is locally terminating, and thus main3 is terminating.

Next we state the soundness of Algorithm 1, and its use for proving termination of a program P. Proving termination of a program can be done, according to Lemma 1, by calling TERMINATES$(S, \emptyset)$ for each scope S.

Theorem 1 (Soundness) *Given a scope S, if* TERMINATES$(S, \emptyset)$ *returns true, then S is locally terminating.*

Corollary 1 (Program termination) *Given a program P and its set of scopes $\mathcal{SS}et$. If for any $S \in \mathcal{SS}et$ the call* TERMINATES$(S, \emptyset)$ *returns true, then P is terminating.*

3.3 Inferring Field-Boundedness

The termination algorithm of this section gives us an automatic technique to infer field-boundedness, i.e., knowing that field f has upper and lower bounds on the values that it can take, however, without specifying the actual bounds. The *upper* (resp. *lower*) bound of a field f is denoted as f^+ (resp. f^-).

Corollary 2 *Consider a field f. If all scopes that reach a point in which f is updated are terminating, then f is bounded.*

4 Cost Analysis

As for termination, the cost of executing a fragment of code can be affected by concurrent interleavings in the loops. Previous work [4] is not able to estimate the cost in these cases. This section proposes new techniques to bound the number of iterations of loops in the presence of concurrent interleavings, which can then be combined, as explained in Sect. 2.4, into an upper-bound on the cost of the corresponding program.

Section 4.1 discusses some necessary background on cost analysis of sequential programs, and states some assumptions on which we rely later; Sect. 4.2 describes the intuition behind our algorithm; Sect. 4.3 describes our algorithm for inferring loop bounds; and Sects. 4.4 and 4.5 describe some further improvements to this algorithm.

4.1 Background on Cost Analysis of Sequential Programs

Our algorithm for computing loop bounds makes use of techniques developed for cost analysis of sequential programs. In a sequential setting, computing a loop bound is done [5] by solving the following two problems: (1) the first one is to bound the number of iterations that a loop can make, in terms of its input; and (2) the second one is to transform this bound into one in terms of the program's input instead of the loop's input. Let us explain this by an example. Consider the following *sequential* C-like program:

```
1  void main(int n) {
2      int j = 2*n;
3      int i=0;
4
5      while (i<j) {
6          i=i+1;
7          m(i);
8      }
9  }
```

```
10 void m(int k) {
11     int l=2*k;
12
13     while (l>0) {
14         l=l-1;
15     }
16 }
17
18
```

We are interested in inferring sequential loop bounds for the loops that start at L125 and L133. Since termination analysis is typically done by synthesizing ranking functions for the different loops, almost any off-the-shelf termination analyzer is able to infer that (1) the loop at L125 can execute at most $\texttt{nat}(j - i)$ iterations, where $\texttt{nat}(v) = \max(v, 0)$ is used to lift negative values to 0; and (2) the loop at L133 can execute at most $\texttt{nat}(l)$ iterations. These bounds, however, are given in terms of the loop's input, and our interest is in ones in terms of the program's input (i.e., the parameter of main). Transforming these bounds into ones in terms of the program's input would result in: (1) $\texttt{nat}(2 * n)$ for the loop at L125, since the maximum value of j is $2 * n$ and the minimum value of i is 0; and (2) $\texttt{nat}(4 * n - 2)$ for the loop at L133, since the maximum value of l is $2 * k$, and the maximum value of k (i.e., i when calling method m) is $2 * n - 1$.

In our algorithm we will make use of a procedure that infers sequential loop bounds (in terms of the loop's input), and a procedure that transforms such bounds into ones in terms of the program's input. The underlying details of these procedures are out of the scope of this paper, we use them as black-box procedures as we describe next.

4.1.1 Sequential Loop Bounds in Terms of Loop's Input.

Recall that the local termination of a scope S, with respect to an assertion C, is analyzed by $\texttt{seq\_termin}(S, C)$ as follows: first the code of S is instrumented with the assertion C at the release points, then all inner scopes are replaced by corresponding summaries and finally the termination of the resulting *sequential code* is analyzed by an off-the-shelf termination analyzer. In the rest of this section, we assume that `seq_termin`, apart from proving termination, also computes a sequential loop bound, denoted NITERS_S, on the number of iterations of this sequential code, in terms of the input to S. Recall that method scopes do not iterate, and thus their corresponding bounds are 1 by definition. In practice, we use the techniques of [5] that are based on linear ranking functions. However, any other technique for inferring such bounds could be used.

Example 7 The following are sequential loop bounds for the different loop scopes of the programs of Figs. 2 and 4:

$$\begin{array}{ll} \text{NITERS}_{S_7} = \texttt{nat}(\mathit{list}) & \text{NITERS}_{S_{89}} = \texttt{nat}(f) \\ \text{NITERS}_{S_{23}} = \texttt{nat}(\mathit{pending}) & \text{NITERS}_{S_{101}} = \texttt{nat}(m) \\ \text{NITERS}_{S_{27}} = \texttt{nat}(\mathit{srvs\_it}) & \text{NITERS}_{S_{113}} = \texttt{nat}(n) \\ \text{NITERS}_{S_{40}} = \texttt{nat}(\mathit{pending}) & \end{array}$$

4.1.2 Sequential Loop Bounds in Terms of Program's Input

Given a sequential loop bound NITERS_S, we let $\texttt{max\_init}(\text{NITERS}_S)$ be an expression that corresponds to transforming NITERS_S into one in terms of the program's input (the parameters of main). Namely, $\texttt{max\_init}(\text{NITERS}_S)$ is an expression that is greater than or equal to NITERS_S whenever the execution reaches the entry point of S. In practice, we use the *maximization* procedure of [5] which is based on the use of linear invariants and parametric linear programming. However, in order to adapt it to our setting, we first replace release points by (1) $f \in [f^-, f^+]$, for each field f that is bounded—see Sect. 3.3—to indicate that f can take any value between f^- and f^+; and (2) $f = *$, for each field that is not bounded, to indicate that it can take any value. It is important to note that $\texttt{max\_init}(\text{NITERS}_S)$ might include also f^- and f^+ (apart from the parameters of main). Note also that this is not the only way to implement `max_init`. We could also use the techniques of [17] which might lead to more precise expressions and eliminate the use of f^- and f^+ as well. However, in this case we would need to modify the work-flow of our algorithm (we will comment on this in Sect. 7).

Example 8 The following are the sequential loop bounds of Example 7, after transforming them to be in terms of the corresponding program's input:

$$\begin{array}{ll} \texttt{max\_init}(\text{NITERS}_{S_7}) = \texttt{nat}(\mathit{tsks}) & \texttt{max\_init}(\text{NITERS}_{S_{89}}) = \texttt{nat}(f^+) \\ \texttt{max\_init}(\text{NITERS}_{S_{23}}) = \texttt{nat}(\mathit{pending}^+) & \texttt{max\_init}(\text{NITERS}_{S_{101}}) = \texttt{nat}(x) \\ \texttt{max\_init}(\text{NITERS}_{S_{27}}) = \texttt{nat}(\mathit{srvs}) & \texttt{max\_init}(\text{NITERS}_{S_{113}}) = \texttt{nat}(y) \\ \texttt{max\_init}(\text{NITERS}_{S_{40}}) = \texttt{nat}(\mathit{pending}^+) & \end{array}$$

4.2 Basic Reasoning

Let us explain the intuition behind our algorithm for inferring loop bounds, using the program of Fig. 4. We assume that we have already applied termination analysis and inferred sequential loop bounds as in Examples 7 and 8.

The loop bounds for S_{101} and S_{113} are straightforward, i.e., they are as the sequential ones $\texttt{nat}(x)$ and $\texttt{nat}(y)$ respectively, since their termination does not depend on any field. The challenge is to infer the loop bound for S_{89}. First note that without interleavings, S_{89} iterates at most $\texttt{nat}(f^+)$ iterations, and let us see how interleaving with S_{101} and S_{113} can affect its number of iterations.

- Suppose that due to an interleaving of S_{89} with S_{101}, the update at L104 is executed, which sets f to some value. Then, in the worst case, when S_{89} resumes it can make $\texttt{nat}(f^+)$ iterations more, if no interleaving occurs again. Thus, we can conclude that an interleaving of S_{89} with L104 can contribute $\texttt{nat}(f^+)$ more iterations to S_{89}. Assuming that NVISITS(L104) is the maximum number of times L104 is visited, then in total it might add $\texttt{nat}(f^+) * \text{NVISITS(L104)}$ iterations to S_{89}. Note that NVISITS(L104) is exactly the loop bound of S_{101}, i.e., $\texttt{nat}(x)$, and thus the above expression equals to $\texttt{nat}(f^+) * \texttt{nat}(x)$. Here we refer to the function NVISITS informally, but its concrete definition will be presented in Algorithm 2.
- Let us see now the effect of the interleaving of S_{89} and S_{113} on the number of iterations of S_{89}. Since the update at L116 does not violate the assertion used to prove the termination of S_{89}, namely $C \equiv \{f' \leq f\}$, we can conclude that it does not affect the number of iterations of S_{89}, and thus can be safely ignored.

To conclude, the loop bound of S_{89} is $\texttt{nat}(f^+)+\texttt{nat}(f^+)*\texttt{nat}(x)$. Here the first $\texttt{nat}(f^+)$ corresponds to the iterations of S_{89} before the first interleaving with S_{101}, and $\texttt{nat}(f^+) * \texttt{nat}(x)$ to the contribution of the interleavings with S_{101}.

Let us formalize the above reasoning. We first fix some notation. We assume that the termination analysis algorithm of Sect. 3 has been applied to all scopes of the program, and that for each scope S we have a corresponding sequential loop bound NITERS_S (computed by `seq_termin`) and its corresponding $\texttt{max\_init}(\text{NITERS}_S)$. The set I computed at L8 of Algorithm 1 is denoted by I_S in order to distinguish it for each scope. We let $\text{NVISITS}(p)$ be an upper-bound function (in terms of the program's input) on the number of visits to program point p in any trace (later we will see that $\text{NVISITS}(p)$ is computed using the loop bounds as well).

Observation 2 *(Loop bound) The loop bound of a scope S is defined as* $\text{LBOUND}_S = \texttt{max\_init}(\text{NITERS}_S) * (rst + 1)$ *where* $rst = \sum_{\{p \mid p \in I_S\}} \text{NVISITS}(p)$.

In the above definition, *rst* represents the number of visits to program points that might change NITERS_S in an arbitrary way.

4.3 Computing Loop Bounds

Our algorithm for computing loop bounds is depicted in Algorithm 2. It consists of two mutually recursive methods: LBOUND for computing a loop bound for a scope S; and NVISITS for computing a bound on the number of visits to a program point p. We assume that termination analysis has been applied, and that for each scope S we have the following (as described before in Observation 2): (1) the set I_S of program points whose instructions might violate the assertion C when proving the sequential termination of S by `seq_termin`; (2) the sequential loop bound NITERS_S and its corresponding $\texttt{max\_init}(\text{NITERS}_S)$.

The function LBOUND receives a scope S for which we want to compute a loop bound and a set of scopes *SSet*, which, as in the case of Algorithm 1, is initially empty and is used to detect cyclic dependencies at L2. At L3, if S is a method scope, it simply returns 1 (since method scopes do not iterate). Otherwise, if S is a loop scope, LBOUND uses Observation 2

Algorithm 2 Bounding the Number of Iterations for Loops with Interleavings

```
1: function LBOUND(S, SSet)
2:   if S ∈ SSet then return false
3:   if S is a method scope then return 1
4:   rst = 0;
5:   for each p ∈ I_S do
6:     rst = rst + NVISITS(p, SSet∪S)
7:   return max_init(NITERS_S) * (rst + 1)
8:
9: function NVISITS(p, SSet)
10:  V_p = 0;
11:  P = reachable_paths(p);
12:  for each ⟨S_1, ..., S_n⟩ in P do
13:    V_aux = 1;
14:    for i = 1 to n do
15:      V_aux = V_aux * LBOUND(S_i, SSet)
16:    V_p = V_p + V_aux
17:  return V_p
```

at L4–L7 to compute the loop bound. The loop at L5 traverses all program points in I_S, and their contribution is added to *rst*. Finally at L7 it returns the loop bound for S as defined in Observation 2. Note that to compute the number of visits to a given program point p, LBOUND uses the function NVISITS that we explain next.

The function NVISITS receives a program point p whose number of visits we want to bound. It also receives a set of scopes *SSet* to detect cyclic dependencies at L2. However, this set is not used directly in NVISITS, it is just passed back to LBOUND when calling it recursively. At L11 we compute a multiset of all scope paths that can reach p. Each path is of the form $\langle S_1, \ldots, S_n \rangle$ where S_1 is the main scope and $p \in S_n$. This can be done using the scopes DAG as follows: we start from the main scope and take all possible paths that lead to the scope to which p belongs. The number of visits to p that is contributed by each path $\langle S_1, \ldots, S_n \rangle$ is

$$\text{LBOUND}_{S_1} * \cdots * \text{LBOUND}_{S_n}$$

since these scopes are nested. This is computed at L13–L15, and accumulated into V_p at L16. This V_p is the total number of visits that is returned at L17.

Example 9 Consider the program of Fig. 2. Let us consider method consumeSync invoked from main3 and compute LBOUND($S_{40}, \emptyset$). Recall that in Example 8 we have computed the corresponding sequential loop bound $\texttt{max\_init}(\text{NITERS}_{S_{40}}) = \texttt{nat}(\mathit{pending}^+)$. Since $I_{S_{40}} = \{L16\}$, we first need to compute NVISITS(L16, $\{S_{40}\}$). At L11 of function NVISITS, we compute the paths that reach L16, which results in $\{\langle S_{\texttt{main3}}, S_{\texttt{addTasks}}, S_7, S_{\texttt{addTask}} \rangle\}$. Then, the loop at L12 of function NVISITS computes the number of visits to L16 by multiplying the results of LBOUND($S_{\texttt{main3}}, \{S_{40}\}$), LBOUND($S_{\texttt{addTasks}}, \{S_{40}\}$), LBOUND($S_7, \{S_{40}\}$) and LBOUND($S_{\texttt{addTask}}, \{S_{40}\}$). The calls that correspond to the scopes $S_{\texttt{main3}}$, $S_{\texttt{addTasks}}$ and $S_{\texttt{addTasks}}$ simply return 1 since they are method scopes. The call LBOUND($S_7, \{S_{40}\}$) returns $\texttt{nat}(\mathit{tsks})$ since $I_{S_{40}} = \emptyset$ (i.e., the loop at L5 ofLBOUND is not executed). Thus, NVISITS($S_{16}, \{S_{40}\}$) returns $\texttt{nat}(\mathit{tsks})$, and then LBOUND($S_{40}, \emptyset$) returns $\texttt{nat}(\mathit{pending}^+) * (1 + \texttt{nat}(\mathit{tsks}))$. Note that this bound is not tight. However, we will improve it in the next section.

The following theorem ensures the soundness of our approach. The proof can be found in Appendix "Proof of Theorem 1".

Theorem 2 (Soundness) *Given a loop scope S, then the call* NITERS$(S, \emptyset)$ *terminates and returns a loop bound for S.*

4.4 Further Improvements: Increment Interleaving

In Observation 2, when a field is updated during an interleaving, we have assumed a worst-case in which it can take any value. However, in practice, updates that increment or decrement a field by some constant value are very common. In this section we present an improvement to handle such cases to obtain more precise bounds. Let us start by an example that explains the intuition behind our idea.

Example 10 Consider the program of Fig. 4, and assume that the update at L116 is f = f + 4 instead of f=f−1. Reasoning as in Observation 2, we would conclude that interleavings of S_{89} with L116 might contribute $\texttt{nat}(f^+) * \texttt{nat}(y)$ iterations to S_{89}. However, since f is incremented by 4 in each visit to L116 we can do better as we show next. Consider scope S_{89}, and let us instrument the release point at L91 with $f - z \leq f' \leq f + z \wedge z \geq 0$ to indicate that f can be incremented or decremented by at most z at this release point. Additionally, note that f is decremented in L92 (f=f−1). Let $\text{NITERS}^{\downarrow}_{S_{89}}$ and $\text{NITERS}^{\uparrow}_{S_{89}}$ be the values of $\text{NITERS}_{S_{89}}$ when starting and completing an iteration of (the instrumented) S respectively. By combining the effects of the release point and the decrement, it is easy to see that

$$\text{NITERS}^{\downarrow}_{S_{89}} - \text{NITERS}^{\uparrow}_{S_{89}} = f' - f \leq z - 1.$$

This means that modifying f by at most z might add at most z iterations to S_{89}. Note that the -1 corresponds to the field decrement in L92 and amounts for the execution of the instrumented iteration. Now since the instruction f=f+4 at L116 can be executed at most $\texttt{nat}(y)$ times (the loop bound of S_{113}), then the total contribution of such updates to the iterations of S_{89} is $4 * \texttt{nat}(y)$, which is more precise than $\texttt{nat}(f^+) * \texttt{nat}(y)$.

Let us present the above idea formally. Let $p \in I_S$ be a program point in which field f is modified (incremented or decremented) by at most the positive constant diff_p. We define S^p as S after instrumenting each release point q that appears in S as follows (1) we add $f - z_q \leq f' \leq f + z_q \wedge z_q \geq 0$ to indicate that f can be incremented or decremented at most by z_q; and (2) for any other field g, we add $g' = *$ to indicate that it might take any value. Let $\text{NITERS}^{\downarrow}_{S}$ and $\text{NITERS}^{\uparrow}_{S}$ refer to the values of NITERS_S when starting and completing an iteration of S^p. We say that $p \in I_S$ is a *bounded update* for S if $\text{NITERS}^{\downarrow}_{S} - \text{NITERS}^{\uparrow}_{S} \leq \sum(z_q) - 1$ holds for S^p. Consider that if there are no interleavings (all z_q are 0) we fall back to the basic situation where the number of iterations decreases at least by one in each iteration $\text{NITERS}^{\downarrow}_{S} \leq \text{NITERS}^{\uparrow}_{S} - 1$. We let $\mathit{inc}(I_S) \subseteq I_S$ be a set of bounded updates, and for each $p \in \mathit{inc}(I_S)$ we let diff_p be the maximum amount by which the corresponding field might be incremented or decremented at program point p. Both $\mathit{inc}(I_S)$ and diff_p can be computed with the help of an SMT solver.

Observation 3 *(Loop bound with constant increments) The loop bound of a scope S is* $\text{LBOUND}_S = \texttt{max\_init}(\text{NITERS}_S) * (1 + \mathit{rst}) + \mathit{inc}$ *where* $\mathit{rst} = \sum_{p \in I_S \setminus \mathit{inc}(I_S)} \text{NVISITS}(p)$ *and* $\mathit{inc} = \sum_{p \in \mathit{inc}(I_S)} \mathit{diff}_p * \text{NVISITS}(p)$.

Modifying Algorithm 2 to use the above observation is done as follows: (i) we insert the following loop immediately before L4

1: $\mathit{inc} = 0$;

2: **for each** $p \in inc(I_S)$ **do**
3: $inc = inc + diff_p * \text{NVISITS}(p, SSet \cup S)$

(ii) at L5 we change I_S by $I_S \setminus inc(I_S)$; and (iii) at L7 we change the return value to $\texttt{max\_init}(\text{NITERS}_S) * (rst + 1) + inc$.

Example 11 Consider the program of Fig. 2, and the call $\text{LBOUND}(S_{40}, \emptyset)$ as developed in Example 9. Recall that $I_{S_{40}} = \{L16\}$. Using the techniques described above, we can show that $inc(I_S) = \{L16\}$ and that $diff_{L16} = 1$ (since we add one element to the list pending at L16). The code presented above (after Observation 3) sets *inc* to $\texttt{nat}(tsks)$, since $\text{NVISITS}(\text{L16}, \{S_{40}\})$ returns $\texttt{nat}(tsks)$ and $diff_{L16} = 1$. The loop L5 of Algorithm 2 does not execute since $I_{S_{40}} \setminus inc(I_{S_{40}}) = \emptyset$ and thus *rst* remains 0. Therefore, the call to $\text{LBOUND}(S_{40}, \emptyset)$ returns $\texttt{nat}(pending^+) + \texttt{nat}(tsks)$ which is more precise than the bound $\texttt{nat}(pending^+) * (1 + \texttt{nat}(tsks))$ obtained in Example 9.

The improvement that we have presented in this section can be generalized. Instead of computing the effect of updating one field f at a time, we could also consider updates to several fields at the same time. This can improve the precision for loops whose termination depends on several fields. Note that the technique would not directly work for scopes that include nested loops with release points in the inner loops. However, only slight modifications are required to make it work for such cases—using the summaries as in `seq_termin`, we can remove inner loops.

4.5 Further Improvements: Visits Versus Atomic Visit

In Observation 3 and its implementation in the modified Algorithm 2 (as described in Sect. 4.4), we classified the program points of I_S into two classes: (A) those that correspond to *bounded updates*, i.e., $inc(I_S)$; and (B) those that correspond to arbitrary updates. In this section, we describe an improvement to be used when computing the number of visits to program points of class (B). In what follows, a program point p means a program point of class (B) above and we assume a given scope S for which we are computing a loop bound.

Intuitively, "a visit to a program point p" was used to model "an interleaving with scope S". For each such interleaving we assumed a worst-case behavior, i.e., that NITERS_S can be set to its maximum value $\texttt{max\_init}(\text{NITERS}_S)$. Now suppose we can guarantee that several visits to p will occur during a single interleaving, then, clearly, it is enough to assume that NITERS_S is set only once to its maximum value. This is because S will not resume before all these updates are performed. For example, consider the program of Fig. 4, and assume that we do not have the **await** instruction at L103. Then, L104 will interleave at most once with L91, and thus it will add at most $\texttt{nat}(f^+)$ iterations to S_1 instead of $\texttt{nat}(x) * \texttt{nat}(f^+)$. Note that such reasoning is not valid for the case for program points of class (A). This is because, in such cases, each visit might increment NITERS_S by a constant d, and thus n visits might increment NITERS_S by $n * d$ independently from whether they were executed during several interleavings or a single one.

We refer to this new notion of number of visits as *atomic number of visits*. Before describing the changes required in our algorithm to support this new notion, we will discuss more elaborated examples in order to understand the different scenarios that we have to account for. Consider the following code snippet

Algorithm 3 Improved version of NVISITS

```
1: function N_ATOMIC_NVISITS(p, RP, SSet)
2:   V_p = 0;
3:   P = reachable_paths(p,RP);
4:   for each ⟨S_1, ..., S_n⟩ in P do
5:     V_aux=1;
6:     mustCount=false;
7:     for i = n to 1 do
8:       if (await y? appears in S_i) || mustCount then
9:         V_aux = V_aux ∗ LBOUND(S_i, SSet)
10:      if S_i is a method scope then
11:        mustCount =true;
12:    V_p = V_p + V_aux
13:  return V_p
```

```
19 // Case 1
20 void p(y,xi) {
21   while (y>0) {
22     x=xi;
23     y=y−1;
24     await z?;
25     while (x>0){
26       x=x−1;
27       f=∗;
28     }
29   }
30 }
31
32
```

```
33 // Case 2
34 void n(y,xi) {
35   while(y>0) {
36     q!m(xi);
37     y−−;
38   }
39 }
40
41 void m(x) {
42   while (x>0) {
43     x=x−1;
44     f=∗;
45   }
46 }
```

which includes two separated cases that we will discuss separately. For Case 1, the field update at L147 appears inside a nested loop and it can be visited $\texttt{nat}(y) * \texttt{nat}(xi)$ times. However, each instance of the innermost loop is executed without releasing the lock. Therefore, each instance of the innermost loop counts as a single visit for other interleaving tasks. The outer loop contains an **await** instruction that might release the lock so we have to count all its iterations. The number of *atomic visits* to L147 would be $\texttt{nat}(y)$. Now let us consider Case 2. Here the number of visits to L164 is $\texttt{nat}(y) * \texttt{nat}(xi)$ as before. In this case neither the inner nor the outer loops contain a release point. However, we have to take the iterations of the outer loop into account because each iteration creates a new task. The execution of these tasks can interleave freely. Consequently, the number of *atomic visits* to L164 is $\texttt{nat}(y)$.

Function N_ATOMIC_NVISITS of Algorithm 3 computes the number of atomic visits to a program point p. In this algorithm, we start by traversing each path $\langle S_1, \ldots, S_n \rangle$ from the innermost scope and we do not add the number of iterations of S_i as long as (1) there are no **await** instructions in S_i; or (2) we have reached a method scope. This is because, if there are no **await** instructions in S_i, then S_i is executed atomically and we only have to count one visit. If we encounter a method scope, each visit to that scope will consist of a newly created task and therefore they do not have to be executed atomically even if there are no release points inside the scope. Incorporating this change in Algorithm 2 amounts to changing the call to NVISITS at L6 by a call to N_ATOMIC_NVISITS. Note that the call to NVISITS in the code after Observation 3, i.e., the modification of Algorithm 2, cannot be replaced by N_ATOMIC_NVISITS since it processes program points of class (A). The correctness of this improvement is justified in Appendix "Soundness of N_ATOMIC_NVISITS".

5 Experimental Evaluation

We have implemented the presented termination and cost analyses and integrated them in the *Static Analyzer for Concurrent Objects* (SACO) tool for analyzing ABS programs—the tool is available at http://costa.ls.fi.upm.es/saco. The termination analysis is launched by enabling the options *Rely guarantee: yes* and *Cost model: Termination*. As the cost analysis is parametric on the cost model used, selecting any other value like *Steps* or *Memory* in the option *Cost model* will launch the cost analysis. Before executing the analyses the user must select one method as the entry point. The output of the termination analysis is the list of reachable scopes from the entry point that are terminating. For the cost analysis the output contains the list of all the reachable methods from the entry point and their cost expressions. This information is shown in the console but it is also integrated in the editor, so the programmer can navigate the code and see the termination information and the cost expressions directly in the methods and scopes.

This section performs an evaluation of the performance and precision of the termination (Sect. 5.1) and cost analyses (Sects. 5.2 and 5.3) in comparison with the analyses presented in [4], which we will call *original* analyses. Regarding the comparison of both cost analyses we will consider two notions of precision: a) the number of methods whose cost expression can be inferred and b) the relation between those cost expressions obtained in order to determine which one is smaller and how large is this difference. We will focus on 4 sets of ABS programs. The first set (BookShop, BoundedBuffer, Chat, DistHT, MailServer and PeerToPeer) are standard ABS programs used to test static analyses in ABS. They model different distributed problems like managing a book shop, a concurrent bounded buffer, a messaging system or a distributed hash table. The second set is Replication System, an industrial case study developed by Fredhopper® that provides search and merchandising IT services to e-Commerce companies. This case study was developed within the HATS project (http://www.hats-project.eu/). The third set contains programs extracted from other papers about static analysis and testing of ABS programs: Fact and CostlyFact from [24] are two versions for computing factorial numbers in a distributed setting; DB from [14] is a simple distributed database protocol; SleepingBarber and Pairs from [13] are classical problems of cooperation in distributed environments. Finally, the fourth set is composed by the running examples of the article: Running1, Running2 and Running3 from the different main methods in Fig. 2; and Interleavings from Fig. 4. The source code for all examples can be found at http://costa.ls.fi.upm.es/saco.

5.1 Termination

Table 1 shows a comparison between the original and the rely-guarantee termination analyses. For every program we show the lines of code[2] (LOC), the number of methods (#m) and the number of scopes (#s). Then we have measured the time (in milliseconds) and the number of scopes (#ts) that can be proved terminating using the original and rely-guarantee termination analyses. For the standard ABS programs both analyses obtain the same results: they can prove all the scopes terminating. The reason of these results is that the standard programs have been developed carefully to avoid loops based on fields, so the rely-guarantee termination analysis does not bring any advantage. It performs slower than the original analysis because it needs to do extra work like computing the MHP pairs between release and field update points. The difference however is not very big: the rely-guarantee analysis is only 1.64–

[2] Lines of code exclude comments and blanks.

Table 1 Comparison of termination analyses on an Intel® i7-4790 at 3.60 GHz with 16GB

ABS program	LOC	#m	#s	Original [4]		Rely-guarantee	
				Time	#ts	Time	#ts
BookShop	227	11	54	278	54	457	54
BoundedBuffer	76	8	17	120	17	219	17
Chat	230	34	45	99	45	312	45
DistHT	126	10	19	81	19	159	19
MailServer	78	9	19	118	19	221	19
PeerToPeer	183	16	44	297	44	696	44
Replication System	1876	143	427	702597	–	56327	424
Fact	28	3	5	21	5	43	5
CostlyFact	29	3	5	20	5	44	5
DB	47	7	11	21	11	48	11
SleepingBarber	52	10	11	31	11	80	11
Pairs	45	6	9	32	9	71	9
Running1	50	6	15	56	15	137	15
Running2	48	6	14	37	12	142	14
Running3	47	6	14	38	12	146	14
Interleavings	48	6	9	38	7	116	9

3.15 times slower. Regarding the programs extracted from other papers the situation is very similar: all the scopes can be proved terminating with both analyses (there are few fields and they do not affect loops) and the rely-guarantee analysis is 2–2.6 times slower. The situation with Replication System is the contrary: the rely-guarantee can analyze the program in about 1 min, but the original termination analysis cannot (it throws a stack overflow error after 12 min of execution). The rely-guarantee analysis obtains a very good result of 424 terminating scopes, only 3 scopes cannot be proved terminating. The improvement in speed is explained because the rely-guarantee analysis works modularly at the level of scopes instead of analyzing the program globally as the original analysis, which leads to the stack overflow error. This benefit is not apparent in smaller programs because it is shadowed by the MHP computation, but as the programs grow, the time required for the MHP analysis becomes insignificant compared to the time of analyzing the program globally. Finally the running examples show a similar behavior. These programs contain loops based on fields and concurrent interleavings, so the rely-guarantee analysis can prove all of them terminating whereas the original analysis cannot prove the termination of those scopes. The exception is Running1, which does not contain any concurrent interleaving, so both analyses obtain the same results. Similarly, the rely-guarantee is about 3 times slower for these programs. Because of their small size, the MHP computation constitutes an important fraction of the total time.

5.2 Cost Analysis

Table 2 shows a comparison between the original and the rely-guarantee cost analyses using *steps* as cost model. For every program we show the lines of code (LOC), the number of methods (#m) and the number of class fields (#f). Then we measure the time (in milliseconds)

Table 2 Comparison of cost analyses on an Intel® i7-4790 at 3.60 GHz with 16GB

ABS program	LOC	#m	#f	Original [4]		Rely-guarantee	
				Time	#bm	Time	#bm
BookShop	227	11	18	340	5	32002	11
BoundedBuffer	76	8	5	153	5	498	8
Chat	230	34	21	127	23	8091	34
DistHT	126	10	3	107	3	419	10
MailServer	78	9	4	142	4	481	9
PeerToPeer	183	16	11	345	7	19321	16
Replication System	1876	143	125	–	–	–	–
Fact	28	3	0	21	3	51	3
CostlyFact	29	3	0	23	3	51	3
DB	47	7	3	23	7	107	7
SleepingBarber	52	10	0	33	10	89	10
Pairs	45	6	2	33	6	114	6
Running1	50	6	1	67	2	201	6
Running2	48	6	1	42	2	170	6
Running3	47	6	1	41	2	192	6
Interleavings	48	6	1	41	4	166	6

and the number of methods (#bm) whose cost expression can be inferred using the original and rely-guarantee cost analyses. The conclusion is that the rely-guarantee cost analysis is slower than the original analysis, but it obtains better results in all programs. For the standard ABS programs, the rely-guarantee cost analysis is 3.3 (BoundedBuffer) to 94 (BookShop) times slower. This difference in time is explained by several reasons. First, the rely-guarantee cost analysis performs a complete termination analysis before starting to infer the cost expressions, since it needs information like the sequential loop bounds or the instructions that may happen in parallel with a scope. Second, it needs to maximize the local loop bounds in terms of the entry arguments (`max_init` function in line 7 of Algorithm 2). This step requires a global analysis of the program, and it becomes harder as the size of the programs or the number of involved fields increase. Therefore we obtain that the slower times are the ones for the bigger programs with a higher number of fields: BookShop, Chat and PeerToPeer. Despite being slower, the rely-guarantee cost analysis obtains cost expressions for all the methods in the programs. The original analysis obtains a smaller ratio of successful cost expressions for the methods (from 30% in DistHT to 68% in Chat), as it fails to infer some local loop bounds or to maximize the cost expressions. Moreover the methods for which the original cost analysis can infer the cost expressions are basically trivial methods like object constructors and other methods without loops. The results of the original analysis can be improved by adding annotations in the programs to assert that the value of a field does not change at a release point or that the value of a field is bounded. These assertions, which must be sound to obtain sound cost expressions, are the ones that the rely-guarantee analysis generates and proves in the termination step. In the Replication System case study both analyses fail to infer cost expressions. This was expected in the original cost analysis, as it is a more complex version of the termination analysis that failed with the case study. For the rely-guarantee cost analysis, the problematic step is the `max_init` function: it needs to analyze the program

globally taking into account all the fields. Since the Replication System is a big program with many fields, this step requires a great amount of time (the analysis was killed after 1 h of execution). In the third set of programs (from Fact to Pairs) the rely-guarantee cost analysis is also 2.2–4.6 times slower but it obtains cost expressions for all the methods. As before, it performs slower as the number of fields increases (DB and Pairs), whereas the time spent by the original analysis is very similar. In these programs the original analysis can also obtain cost expressions for all the methods. The explanation is that in these programs there are few fields and they do not affect any loop, so the improvements in the rely-guarantee analysis are not needed. For the running examples, we observe the same behavior as in the standard programs: the rely-guarantee is slower but obtains cost expressions for all the methods, whereas the original analysis is faster but fails to infer several cost expressions. In this case the rely-guarantee is only 3 to 4.7 times slower (for Running1 and Running3 respectively), but the differences are small because the size and the number of fields are very similar. Regarding precision, the original cost analysis infers cost expressions for 41.25% of the methods on average. As before, it succeeds with simple methods like constructors but fails when analyzing loops with interleavings.

5.3 Precision of the Cost Expressions

The rely-guarantee analysis does not only infer cost expressions for more methods, but it also obtains more precise (smaller) cost expressions than the original analysis. We cannot check the precision of the obtained upper bounds w.r.t. the actual cost of running these programs, as currently there is no tool for computing this cost for ABS programs. Therefore to prove this claim, we have compared the cost expressions obtained by both analyses (using *steps* as cost model). Cost expressions are functions with arguments that are method parameters or class fields, so in general its comparison is not straightforward. The simplest case appears when the obtained cost expressions are constant numbers. This happens for example in BoundedBuffer for the method BoundedBuffer.remove, where both analyses obtain an upper bound of 17. This method does not receive any parameter and simply executes a sequence of instructions without any loop, so both analyses can obtain its cost easily. Another case is when the obtained cost expressions are functions depending on different arguments, so its comparison is not possible. An example of this situation is the method A.m2 in Interleavings (method m2 in Fig. 4): the original analysis infers the upper bound $6 + 11 * \texttt{nat}(m)$, based on the method parameter m, but the rely-guarantee analysis infers $14 + 3 * \texttt{nat}(x)$, based on the main parameter x—due to the maximization procedure presented in Sect. 4. These parameters are related, as the possible values of m depend on x, but in general these relations are complex and prevent the comparison of the upper bounds. On the other hand, even when both cost expressions are defined on the same arguments, its comparison is not direct. Let us consider the method Main.main in program Pairs: the original analysis obtains the cost expression $12 + 32 * \texttt{nat}(main\_n)$ and the rely-guarantee analysis $40 + 2 * \texttt{nat}(main\_n)$. Both expressions are defined on the parameter *main_n*, but no function is smaller for all possible inputs. For values of $main\_n \in (-\infty, 0]$ the expression $12 + 32 * \texttt{nat}(main\_n)$ is smaller, whereas for values greater than 0 the expression $40 + 2 * \texttt{nat}(main\_n)$ is clearly smaller. Notice that this situation is simple (one parameter, only addition and multiplication) but general cost expressions combine many parameters and a variety of operators, so even its asymptotic comparison is a complex problem.

We have applied two alternatives to compare cost expressions defined on the same arguments. The first one is the technique presented in [6], which exploits the syntactic properties of the cost expressions to prove if one is smaller than or equal to the other for all input values

of interest. This technique is fully automatic and it is implemented,[3] so we have been able to integrate it in our benchmarks. As expected, when comparing $12 + 32 * \texttt{nat}(main\_n)$ and $40 + 2 * \texttt{nat}(main\_n)$ using the tool in [6], the result is that no cost expression is less than or equal to the other. A different approach for comparing cost expressions—applied in papers about resource analysis like [7–9]—is evaluating them several times with suitable random values for their parameters. Combining all these values, we obtain an average ratio that approximates quantitatively the relation between both cost expressions. Applying this technique, the calculated average ratio between $40+2*\texttt{nat}(main\_n)$ and $12+32*\texttt{nat}(main\_n)$ is 0.07, i.e., the cost expression obtained by the rely-guarantee is, on average, 7% of the original cost expression. This result is expectable since for many values of *main_n*, the multiplication has more weight than the constant added, and the ratio between the factors is $\frac{2}{32} = 0.0625 \approx 0.07$.

Table 3 summarizes the comparison of the cost expressions obtained by the rely-guarantee and original analyses for all the programs. In order to maximize the number of cost expressions that can be compared, we have included annotations in some programs with information about class fields so that the original cost analysis could infer more cost expressions—annotated programs are marked in Table 3 with a subscript like $\mathsf{BookShop}_a$ or $\mathsf{Running1}_a$. As mentioned before, these annotations state that the value of some fields is bounded and that, at some **await** instructions, the value of a field does not change. Thanks to these annotations, the original analysis obtains cost expressions for more methods, but the programmer must include these annotations manually in the program, so the results of the original analysis are only valid if these properties are true in every program execution. We remark that the rely-guarantee analysis does not need any annotation since it proves these properties before computing the cost expressions.

For each program we have executed the original and the rely-guarantee analyses. The column *Original* in Table 3 contains the time (in milliseconds) needed to analyze the program using the original analysis and the number of methods whose cost expression is obtained (#bm). The details for the rely-guarantee analysis are not shown because their values are the same as in Table 2—recall that annotations are ignored, so they do not affect the total time or the obtained cost expressions. Regarding the original analysis, it needs more time to process annotated programs than its initial versions (from 102% of the initial time in Chat to 368% in PeerToPeer). This increment in the total time is explained because without annotations, the original analysis failed prematurely for some methods (it was unable to maximize or find a ranking function) but using the extra information from the annotations it can proceed. Notice that in all the programs but in $\mathsf{Interleavings}_a$, the number of methods with cost expression has increased, and in some cases the original cost analysis obtains cost expressions for all the methods in the annotated program: $\mathsf{BoundedBuffer}_a$, DistHT_a, $\mathsf{MailServer}_a$, $\mathsf{PeerToPeer}_a$, $\mathsf{Running1}_a$ and $\mathsf{Running2}_a$. The column *Rely-Guarantee vs. original* in Table 3 compares the cost expressions inferred by both analyses. For each program it shows the number of methods whose cost expressions are the same constant number (#const) and the number of methods whose cost expressions cannot be compared because they depend on different parameters (#diff). Then for each method whose cost expressions depend on the same parameters, the table shows the method name and the average ratio $\frac{rely-guarantee}{original}$ obtained by evaluating both cost expressions with random values.[4] Notice that in addition to computing the average ratio, we have applied the technique presented in [6] to qualitatively compare these cost expressions, but for all the methods the technique concluded that no

[3] http://costa.ls.fi.upm.es/comparator.

[4] For every comparable method, we computed 200 ratios and compute their average.

Table 3 Cost bounds comparison for the two considered cost analyses on an Intel® i7-4790 at 3.60 GHz with 16GB

ABS program	Original [4]		Rely-Guarantee versus original [4]			
	Time	#bm	#Const	#Diff	Method name	Avg. ratio
BookShop$_a$	458	9	5	3	AgentImp.free	0.22
BoundedBuffer$_a$	166	8	4	2	BoundedBuffer.append	0.23
					Main.main	10^{-4}
Chat$_a$	129	31	22	0	ButtonImpl.registerListener	0.22
					ClientGUIImpl.init2	0.25
					ClientImpl.getGUI	0.28
					ServerImpl.sessionClosed	0.35
					SessionImpl.close	0.36
					ClientImpl.receive	0.38
					SessionImpl.init2	0.37
					ServerImpl.connect	0.38
					ClientImpl.start	0.39
DistHT$_a$	241	10	3	5	Node.getData	0.30
					Node.putData	0.60
MailServer$_a$	173	9	4	0	AddressBookImp.addUser	0.67
					MailServerImp.addUser	0.68
					AddressBookImp.getUserAddress	0.29
					MailServerImp.notify	0.17
					Main.main	6×10^{-4}

Table 3 continued

ABS Program	Original [4]		Rely-Guarantee versus original [4]			
	Time	#bm	#Const	#Diff	Method name	Avg. ratio
PeerToPeer$_a$	1568	16	6	7	DataBaseImpl.listFiles	0.43
					Node.enquire	0.43
					DataBaseImpl.getFile	0.29
Fact	21	3	1	1	Main.main	0.63
CostlyFact	23	3	1	1	Main.main	0.51
DB	23	7	7	0		
SleepingBarber	33	10	10	0		
Pairs	33	6	5	0	Main.main	0.07
Running1$_a$	76	6	2	2	TaskQueue.addTask	0.22
					Main.main	5×10^{-3}
Running2$_a$	57	6	2	1	TaskQueue.addTask	0.22
					TaskQueue.consumeSync	0.21
					Main.main	0.02
Running3$_a$	57	4	2	1	TaskQueue.addTask	0.24
Interleavings$_a$	62	4	2	2		
Total		132	76	25	31	

one was smaller than or equal to the other, so we have not included this result in the table. Regarding the average ratio, we can observe that the cost expressions obtained by the rely-guarantee analysis are smaller in all the cases, with an average ratio ranging from 0.2 to 0.7 approximately. There are some extreme cases where the average ratio is very small, namely for the Main.main methods in $\mathsf{BoundedBuffer}_a$, $\mathsf{MailServer}_a$, $\mathsf{Running1}_a$ and, moderately, in Pairs_a. The reason is that the cost expression for the Main.main methods (the entry point of the programs) combines the cost of all the reachable methods, so it accumulates all the inaccuracies. This behavior does not happen in the Main.main methods of Fact or CostlyFact since they have a small number of methods (only 3) and their Main.main methods are simple: they just create a new object and invoke one method on it. Therefore their cost expressions do not accumulate as many inaccuracies as in the other programs. As a final remark, notice that out of the total of 132 cost expressions that could potentially be compared, only 25 ($\approx$ 19%) are defined on different arguments and therefore cannot be compared. The majority of the cost expressions (76, $\approx$ 58%) are constants, and 31 ($\approx$ 23%) have been effectively compared using the comparator proposed in [6] and computing the average ratio.

5.4 Conclusion

In this section we have showed, by means of different programs, that the presented rely-guarantee analyses obtain better results than the original analyses in [4]. Regarding termination, the rely-guarantee analysis obtains a greater number of terminating scopes in the presence of interleaving involving class fields. Similarly, the rely-guarantee cost analysis obtains cost expressions for more methods and does not need any manual annotation from the programmer. Moreover, even when these annotations are inserted, the rely-guarantee cost analysis obtains more precise (smaller) cost expressions than the original cost analysis.

Regarding scalability, the largest program we have analyzed has 1876 lines of code, and it is an industrial case study. As we have shown in our experiments, termination analysis successfully finishes, as it works modularly. However, for cost, we are not able to complete the analysis. It is indeed unmanageable since some parts require a whole-program analysis. It is also worth noting that, since we only use immutable data structures, we do not need a heap analysis and there is no loss of precision in this regard.

6 Related Work

Existing methods for proving termination of thread-based programs also apply a rely-guarantee or assume-guarantee style of reasoning [19,20,29]. These methods consider every thread in isolation under assumptions on its environment, thus avoiding to reason about thread interactions directly. Applying this technique to our concurrent setting could be done by assuming a property of the second object while proving the property of the first object, and then assuming the recently proved property of the first object when proving the assumed property of the second object. Although we make assumptions and then prove them, our assumptions are of a different kind, i.e., namely they are assumptions on finiteness of data, no matter on which thread (or object) they are executed. This point makes our work fundamentally different from [19]. We can still apply our method in the presence of dynamically created objects and the number of concurrency units does not need to be known a priori as in [19].

In parallel with our work [11], Kupriyanov and Finkbeiner [28] proposed a termination analysis technique for multi-threaded programs that uses a finiteness assumption similar to

ours. Roughly, their analysis maintains a set of potential infinite runs (represented by sets of transitions) that are ruled out incrementally. When considering a set of transitions, if a termination witness is found (i.e., a ranking function) then it cannot represent an infinite run unless this witness is violated infinitely often, which means that a new potential infinite run is generated. This, in principle, is as our finiteness assumption: in our case scopes correspond to potential infinite runs, and once a scope is proved locally terminating the recursive call in our Algorithm 1 corresponds to checking new potential infinite runs that might interfere with the local termination witness. Note that their work is presented in the context of termination analysis while we have extended our method to infer costs as well. It is not clear how their work can be extended to infer cost bounds as well.

The finiteness assumption was also used by Gotsman el al. [25] to prove properties (lock-freedom, wait-freedom, etc.) of non-blocking algorithms. In particular, it is used to prove that a loop inside an operation procedure (e.g., Push or Pop in a non-blocking stack implementation) terminates by requiring that the operation is used finitely often by the program threads (i.e., those threads terminate as well). This, in principle, is similar to our finiteness assumption, however, in our case we do not make this distinction between operations and code that uses them.

As regards the bounds on loop iterations, to the best of our knowledge, there are no other works that have attempted to infer those bounds for loops with concurrent interleavings before. There are several techniques [15,17,18,23,31,32,34] for inferring complex loop bounds for (sequential) programs. Our basic termination component could benefit from these techniques. Moreover, in principle, a concurrent program could be translated to a transition system that simulates all possible interleavings, which then would allow using these techniques for inferring bounds on loops with concurrent interleaving. However, we expect such translation to be far more complicated than our techniques.

Finally, as in other kinds of analyses, by making the analysis *object-sensitive* (i.e., by distinguishing between different objects of the same class) we can achieve further precision. For instance, if we add to main3 the following two instructions TaskQueue q1 = new TaskQueue(); q1!addTasks();. When obtaining the cost of consumeSync as we did in Example 9, we will have to consider $\text{NVISITS}(S_{16}, \{S_{40}\})$. Without distinguishing q and q1, we will obtain two paths that reach program point 16. One path for each call to addTasks. Consequently, the resulting cost will be twice as high. On the contrary, if we distinguish the program point 16 executed in object q and object q1, $\text{NVISITS}(S_{16:q}, \{S_{40:q}\})$ will find only one path and therefore compute a smaller value.

7 Conclusions and Future Work

Concurrency adds further difficulty when attempting to prove program termination and inferring resource consumption. The problem is that the analysis must consider all possible interactions between concurrently executing objects. This is challenging because processes interact in subtle ways through fields and future variables. We have proposed novel techniques to prove termination and inferring upper bounds on the number of iterations of loops with such concurrent interleavings. Our analysis benefits from an existing MHP analysis to achieve further precision [10].

One of the main directions for further research is the improvement of the may-happen-in-parallel analysis, as any precision gain in such analysis directly achieves a precision gain in our method. Recent work [12] has investigated the extension to consider inter-procedural

synchronization in which future variables can be passed as method parameters and thus it is possible to await for the termination of an asynchronous call outside the scope in which it has been spawned. This enhanced analysis can be used directly within our method. As part of our future work, we want to study the extension of the MHP analysis to allow synchronization on boolean conditions. This allows further expressivity but it is known to pose challenges in static analysis. This is because it is difficult to automatically infer to which parts of the program the await instruction synchronizes.

In this work, we have presented an approach limited to non-recursive programs. First, note that the restriction to non-recursive programs is only in the imperative part of the language, the functional part has only recursion (but it is a sequential code), and our sequential termination analysis backend already handles sequential recursive programs. Second, this restriction can easily be lifted for recursive methods (in the concurrent part) that execute their recursive calls in the same object. It only requires modifying the notion of scopes a bit to include mutually recursive methods into a single scope and maintain the containment relationship among scopes ($\rightarrow$) presented in Sect. 2.4 acyclic.

If a method contains recursive calls that might be executed in different objects, our approach is not applicable. For instance, if we have a method m with a recursive call x!m() such that $x \neq this$. Then, the fields in two consecutive recursive calls might be completely different variables (belonging to different objects). Therefore, we cannot establish assumptions among their values. Note that we might still be able to prove termination (and obtain a bound) without establishing any assumptions as long as it does not depend on the values of the fields.

In Sect. 4.1 we have commented on using the techniques of [17] for computing `max_init`(NITERS$_S$). Moreover, we claimed that it leads to more precise bounds. Adapting these techniques to our concurrent setting, however, is not straightforward. First, the underlying algorithms of [17] need to be modified when computing dependencies between program variables, since the value of a field f at a release point might depend on a modification in another method. Second, we also need to change the work-flow of our algorithm to first compute loop bounds using the sequential ones (i.e., use NITERS$_S$ instead of `max_init`(NITERS$_S$) in Algorithm 2), and only afterwards rewrite them to be in terms of the input. This is because the key idea of [17] is to use the loop bounds to compute size bounds (which is all what we need to transform bounds to be in terms of the input). We leave this research direction for future work.

Acknowledgements Funding was provided by Seventh Framework Programme (EU) (Grant no. FP7-ICT-610582), Ministerio de Economía y Competitividad (ES) (Grant nos. TIN2012-38137, TIN2015-69175-C4-2-R), and Comunidad de Madrid (ES) (Grant no. S2013/ICE-3006).

Appendix: Proofs of Theorems

We base our proofs on traces generated by the semantics rules and the behavior of the loop counters described in the definition of local termination and local loop bound. First, we will define some auxiliary notions.

Given a trace, we denote $S_{x:t}$ a scope S_x executing in a task t whose auxiliary counter is $i_{x,t}$. Also we can map each execution step $St_i \rightarrow^{b_i} St_{i+1}$ to the task t in which b_i is executed with $TaskOf(b_i) = t$. As stated in Sect. 2.4, an instruction b_i can only belong to one scope but it can appear inside several scopes. A typical example is an instruction inside a nested loop. $b_i \in S$ denotes that b_i belongs to S and $b_i \sqsubset S$ denotes that b_i appears inside S.

A t-interleaving trace represents a trace fragment where all the execution steps are performed in tasks different from t

Definition 3 *(t-interleaving trace)* A trace $tr^{lv} \equiv St_1 \rightarrow^{b_1} \ldots \rightarrow^{b_{n-1}} St_n \rightarrow^{b_n} \ldots$ is interleaving with task t if for all b_i such that $1 \leq i \leq n$, $TaskOf(b_i) \neq t$.

Definition 4 *(C valid in trace tr)* A shared-memory-assertion C is valid in a trace $tr \equiv St_1 \rightarrow^{b_1} \ldots \rightarrow^{b_{n-1}} St_n$ with respect to an object o, if for all constraints $f' \bowtie f \in C$, $ob(o, a, lk) \in St_1$, $ob(o, a', lk') \in St_n$ and $a'(f) \bowtie a(f)$.

Next we define traces that represent a single iteration of a loop scope in a task $S_{x:t}$:

Definition 5 *($S_{x:t}$ iteration trace)* A trace $tr^{it} \equiv St_1 \rightarrow^{b_1} \ldots \rightarrow^{b_{n-1}} St_n \rightarrow^{b_n} \ldots$ is a iteration trace of $S_{x:t}$ if:

- for all the steps $St_i \rightarrow^{b} St_{i+1}$ either $TaskOf(b_i) \neq t$ or $b_i \sqsubset S_x$. This implies that the loop exit is not taken and $i_{x,t}$ is not reset but there can be t-interleaving traces within tr^{it}
- $i_{x,t}$ is only incremented in b_1 and thus it can represent at most one iteration of $S_{x:t}$.

An iteration trace tr^{it} of $S_{x:t}$ can be infinite. This is because it might contain infinite t-interleaving traces or because S_x contains other loop scopes that can iterate indefinitely.

We say that a finite tr^{it} is *well-behaving* with respect to C if C is valid in all its maximal t-interleaving subtraces. Finally, we define traces that represent a complete execution of a loop scope $S_{x:t}$ without resetting the counter $i_{x,t}$:

Definition 6 *($S_{x:t}$ execution trace)* A trace $tr^{ex} \equiv St_1 \rightarrow^{b_1} \ldots \rightarrow^{b_{n-1}} St_n \rightarrow^{b_n} \ldots$ is an execution trace of $S_{x:t}$ if it can be expressed as a (possibly infinite) sequence of iteration traces tr^{it} and $i_{x,t}$ is 0 in St_1.

In addition, our proof relies on the soundness of the `seq_termin` algorithm [4] and the *may-happen-in-parallel* (MHP) analysis [10] that we state below.

Soundness of `seq_termin` guarantees universal termination of a scope provided the interleaving scopes do not violate the constraint set C. We can define the soundness of `seq_termin` using these notions:

Definition 7 *(Soundness of* `seq_termin`*)* If `seq_termin`$(S_x, C) = true$, then for any task $tsk(t, m, o, l, s) \in St_a$ and trace $tr^{w} \equiv St_a \rightarrow^{b_a} \ldots \rightarrow^{b_{n-1}} St_n \rightarrow^{b_n} \ldots$ that can be expressed as a sequence of *well-behaved* iteration traces of $S_{x:t}$, the number of execution steps that increments $i_{x,t}$ (and the number of iteration subtraces) is finite ($sup(tr, i_{x,t})$ is defined).

The soundness of the `MHP_pairs` analysis states that it overapproximates the set of program points that can happen in parallel, i.e., if two program can be executed in some state St of the trace, then both points will be related in the results of `MHP_pairs`.

Definition 8 *(Soundness of MHP)* Consider a set of program points RP and `MHP_pairs`$(RP) = MP$. If $s_1 \in RP$ and for some reachable state St we have that $tsk(t_1, m_1, o_1, l_1, s_1; s'_1)$ and $tsk(t_2, m_2, o_2, l_2, s_2; s'_2)$ are two tasks available in St, then $s_2 \in MP$.

Proof of Theorem 1

We re-state Theorem 1 using the definition of locally terminating scope:

Theorem 3 (Soundness of TERMINATES(S, L)) *If* TERMINATES$(S_x, L) = true$*, for any task t and any trace tr there exists* $n_{t,tr} \in \mathbb{N}$ *such that* $sup(tr, i_{x,tr}) \leq n_{t,tr}$

Proof For any trace tr we consider any execution trace tr^{ex} of $S_{x:t}$ (for any t) that appear in tr. Then, S_x is locally terminating if $i_{x,t}$ is incremented a finite number of times in all possible tr^{ex}. This is equivalent to proving that tr^{ex} contains a finite number of iteration subtraces (each iteration subtrace increments $i_{x,t}$ once).

We can partition tr^{ex} into an alternative sequence of subtraces ${tr^w}_j$ and ${tr^b}_i$, where ${tr^w}_j$ are maximal subtraces that satisfy the restrictions in Definition 7 (a sequence of well-behaving iteration traces) and ${tr^b}_i$ is a sequence of iteration traces that are not well-behaved. An iteration trace is not well-behaved because it is either infinite or it contains t-interleaving traces that violate C.

`seq_termin`$(S_x, C) = true$ guarantees that every ${tr^w}_j$ increments $i_{x,t}$ a finite number of times. In order to prove that the total number of increments are finite, we need to prove that the number of ${tr^w}_j$ is finite and the number of iterations in all ${tr^b}_i$ (the iterations that are not well-behaving) is also finite. Because we consider maximal subtraces, we have that between two consecutive ${tr^w}_j$ there must be at least one ${tr^b}_i$ that contains at least one iteration that is not well-behaving. Therefore, it is enough to prove that the number of iteration traces that are not well-behaving is finite.

Recall that if an iteration trace is not well-behaving, it must be either infinite or contain at least one t-interleaving trace where C is not valid. If we have that ${tr^{it}}_j$ is infinite, there cannot be other iteration traces after ${tr^{it}}_j$ and we can trivially conclude that the number of total iterations is finite (it is j). If ${tr^{it}}_j$ contains at least one t-interleaving trace tr^{lv} where C is not valid, there must be at least one execution step $St_i \rightarrow^p St_{i+1}$ that makes a constraint in C invalid, and p must be a field assignment. Because ${tr^{it}}_j$ is an iteration trace of $S_{x:t}$, we have that $tsk(t, m, o, l_1, s_1 : s) \in St_i$ and s_1 must be a release point with $s_1 \sqsubset S_x$. This is because if p can perform a field assignment in object o, it has to be executed in a task t' that belongs to the same object o and have the object's lock. In order to take the lock, it must have been released by the last instruction before the interleaving trace. We have that $p \in I = MP(RP)$ where RP are the release points that appear in S_x. Lines 6–8 approximate this set of points. The soundness of I is given by the soundness of the MHP analysis.

If the number of times each p is executed is finite, then the number of subtraces ${tr^{it}}_j$ that are not well-behaving is also finite and S_x is locally terminating. This is guaranteed by proving that every scope that can reach p is locally terminating through the recursive calls to TERMINATES. □

Proof of Theorem 2

We have to prove LBOUND computes a valid local bound for any scope S. The bounds of method scopes are trivial so in what follows we only consider bounds of loop scopes. Our proof relies on the soundness of the sequential loop bound NITERS$_S$ computed by `seq_termin` and the soundness of the `max_init` procedure that we state below:

Definition 9 *(Soundness of* `seq_termin` *sequential loop bound)* If `seq_termin`(S_x, C) returns a function NITERS$_S$, then for any task $tsk(t, m, o, l, s) \in St_e$ and partial trace $tr^w \equiv St_e \rightarrow^{b_e} \ldots \rightarrow^{b_{n-1}} St_n \rightarrow^{b_n} \ldots$ that can be expressed as a sequence of *well-behaved* iteration traces of $S_{x:t}$, the number of execution steps that increments $i_{x,t}$ (and the number of iteration subtraces) is finite and smaller or equal that NITERS$_S$ evaluated in St_e.

Definition 10 *(Soundness of* `max_init`*)* For any partial trace $tr \equiv St_0 \rightarrow^0 \ldots \rightarrow^{b_{e-1}} St_e$ where $tsk(t, m, o, l, s), ob(o, a, t) \in St_e, s$ is the loop scope $S_x, tsk(0, main, 0, l_0, s) \in St_0$ and $\bar{f}^+$ and $\bar{f}^-$ are the upper and lower bounds of the fields of object o. We have: $\text{NITERS}_S(l, a) \leq \texttt{max\_init}(\text{NITERS}_S)(l_0, \bar{f}^+, \bar{f}^-)$.

The implementation of LBOUND corresponds directly with the Observation 2. Therefore, it is enough to proof that the Observation 2 is sound. Below, we re-state the Observation 2 using the definition of loop bound and prove its soundness.

Observation 4 *(Loop bound (expanded)) For any task t and any trace tr,* $sup(i_{x,t}) \leq \text{LBOUND}_S$ *where* LBOUND_S *is defined as* $\text{LBOUND}_S = \texttt{max\_init}(\text{NITERS}_S) * (rst + 1)$ *where* $rst = \sum_{\{p|p \in I_S\}} \text{NVISITS}(p)$.

Proof Similarly to the proof of termination, we consider arbitrary execution subtraces tr^{ex} of S partitioned into an alternative sequence of subtraces $tr^w{}_j$ and $tr^b{}_i$, where $tr^w{}_j$ are maximal subtraces that satisfy the restrictions defined in Definition 7 (a sequence of well-behaving iteration traces) and $tr^b{}_i$ is a sequence of iteration traces that are not well-behaved.

Combining the soundness of the bound returned by `seq_termin` and of `max_init`, we know that each $tr^w{}_j$ increments $i_{x,t}$ at most $\texttt{max\_init}(\text{NITERS}_S)$. However, if $tr^w{}_j$ is not at the end of tr^{ex} it must be possible to execute at least one iteration after $tr^w{}_j$. In that case $tr^w{}_j$ has at most $\texttt{max\_init}(\text{NITERS}_S) - 1$ iterations.

Here again, we have that between two consecutive $tr^w{}_j$ there must be at least one finite $tr^b{}_i$ that contains at least one iteration that is not well-behaving. In addition, we can have at most one infinite not well-behaving iteration trace at the end. Let rst be the number of finite iterations that are not well-behaving, the number of $tr^w{}_j$ subtraces is at most $rst + 1$ where only one of them can be at the end of tr^{ex}. The maximal number of iterations in tr^{ex} is:

- $(\texttt{max\_init}(\text{NITERS}_S) - 1) * (rst) + \texttt{max\_init}(\text{NITERS}_S) + rst$ if tr^{ex} ends with a $tr^w{}_j$ and
- $(\texttt{max\_init}(\text{NITERS}_S) - 1) * (rst + 1) + rst + 1$ if tr^{ex} ends with an infinite not well-behaving iteration trace.

In these expressions (1) $(\texttt{max\_init}(\text{NITERS}_S) - 1) * (rst)$ and $(\texttt{max\_init}(\text{NITERS}_S) - 1) * (rst)$ account for the iterations in all $tr^w{}_j$ that are not at the end of tr^{ex}; (2) rst for the finite iterations that are not well-behaved; and (3) $+1$ corresponds to an infinite not well-behaving iteration trace in the second expression. These two expressions are equivalent to the one in the Observation 4

We can use the same reasoning used in the proof of Theorem 1 to conclude that each finite tr^{it} that is not well-behaving must contain at least one step $St_i \rightarrow^p St_{i+1}$ such that $p \in I$ that makes C invalid. Recall that I is set of field assignments that might happpen in parallel with the release points of S_x. Therefore $rst \leq \sum_{\{p|p \in I_S\}} \text{NVISITS}(p)$.

Finally, $\text{NVISITS}(p)$ corresponds to obtaining the cost of the program $\text{UB}(S_{main})$ given a cost model that assigns $\mathcal{M}(p) = 1$ and 0 to any other instruction. It is inmediate to see that the implementation of $\text{NVISITS}(p)$ corresponds to the definition of $\text{UB}(S_{main})$ with the given cost model in Sec.2.4. □

Soundness of Observation 3

Proof Consider an execution trace tr^{ex} of a loop scope S partitioned into tr^w and tr^b subtraces, but this time we distinguish between the tr^b that contain only interleavings with points in $inc(I_S)$, denoted tr^{inc}, and the ones that might interleave with other points in $I_S \setminus inc(I_S)$

as well, denoted tr^{rst}. We denote *rst* the number of finite iterations that are not well-behaved in tr^{rst}, which is at most $\sum_{\{p|p\in I_S\setminus inc(I_S)\}}$ NVISITS(p). We have at most *rst* tr^{rst} subtraces (if each tr^{rst} contains a single iteration) that split the execution trace in at most $(rst+1)$ tr^{no-rst} subtraces. Each tr^{no-rst} is a sequence of tr^{w} separated by a tr^{inc}.

We know that each tr^{w} has at most NITERS$_S$ iterations expressed in terms of the initial variables of tr^{w} (soundness of `seq_termin`) but also we have that NITERS$_S(\overline{x_i})$ − NITERS$_S(\overline{x_f})$ is an upper bound on the number of iterations of tr^{w} where $\overline{x_i}$ and $\overline{x_f}$ are the variables at the first and last state of tr^{w}. This is because each well-behaved iteration is guaranteed to decrease NITERS$_S(\overline{x_i})$ at least 1 (and decrease at least n in n iterations). Given one of these subtraces tr^{no-rst}[5] $\equiv tr^{w}{}_1 tr^{inc}{}_1 tr^{w}{}_2 \ldots tr^{inc}{}_{n-1} tr^{w}{}_n$, we have that the cost of each $tr^{w}{}_j$ is NITERS$_S(\overline{x_{i:j}})$ − NITERS$_S(\overline{x_{f:j}})$ (where $\overline{x_{i:j}}$ and $\overline{x_{f:j}}$ are the variables of the first and last state of $tr^{w}{}_j$). We have checked that the effect of one iteration of tr^{inc} on NITERS$_S$ is $(\sum z_q) - 1$. Let it_j be the number of iterations in $tr^{inc}{}_j$, the effect of $tr^{inc}{}_j$ is NITERS$_S(\overline{x_{i:j+1}}) \leq$ NITERS$_S(\overline{x_{f:j}}) + \sum_{k=1}^{it_j}((\sum z_{q:k}) - 1)$. We have that the cost of all the $tr^{w}{}_1$ in the sequence tr^{no-rst} is at most $\sum_{j=1}^{n}$ NITERS$_S(\overline{x_{i:j}})$ − NITERS$_S(\overline{x_{f:j}})$ Then, we express each NITERS$_S(\overline{x_{i:j}})$ (for $j > 1$) in terms of NITERS$_S(\overline{x_{f:j-1}})$ and we cancel the positive and negative summands obtaining NITERS$_S(\overline{x_{i:1}})$ − NITERS$_S(\overline{x_{f:n}})$ + *extra* where $extra = \sum_{j=1}^{n-1}\sum_{k=1}^{it_j}(\sum z_{q:k}) - 1$. To complete the cost of the sequence tr^{no-rst}, we add 1 for each iteration in tr^{inc} subtraces: $\sum_{j=1}^{n-1}\sum_{k=1}^{it_j} 1$ which we can cancel with the -1 appearing in *extra*. As a result we have $Cost(tr^{no-rst})$ = NITERS$_S(\overline{x_{i:1}})$ − NITERS$_S(\overline{x_{f:n}})$ + $extra'$ where $extra' = \sum_{j=1}^{n-1}\sum_{k=1}^{it_j}\sum z_{q:k}$.

We can maximize NITERS$_S(\overline{x_{i:1}})$ and substitute NITERS$_S(\overline{x_{f:n}})$ by 0 if tr^{no-rst} is at the end of the execution or by 1 if there are further iterations. Then we denote *inc* the sum of all $extra'$ for every tr^{no-rst} subtrace. The cost of the complete tr^{ex} is : (`max_init`(NITERS$_S$) − 1) $*$ *rst* + *rst* + `max_init`(NITERS$_S$) + *inc* which corresponds to the expression given in Observation 3.

Finally, it is left to justify that *inc* is bounded by $\sum_{p\in inc(I_S)} diff_p *$ NVISITS(p). *inc* represents the sum of all field modifications z_q that occur in interleavings of tr^{ex} where C is not valid and the points in $I_S \setminus inc(I_S)$ are not visited. Consequently, the points in $inc(I_S)$ are the only ones that can contribute to *inc* and they can contribute at most $diff_p$ per program point visit. Therefore $\sum_{p\in inc(I_S)} diff_p *$ NVISITS(p) is a sound approximation of *inc*. □

Soundness of N_ATOMIC_NVISITS

In the previous section, we argued that rst is the number of finite iterations tr^{it} that are not well-behaving and belong to a tr^{rst}. Each not well-behaving tr^{it} in a tr^{rst} must contain at least one t-interleaving trace tr^{lv} with at least one step $St_i \rightarrow^{p} St_{i+1}$ such that $p \in I \setminus inc(I)$ that makes C invalid.

Instruction p is executed in a task $t' \neq t$ such that t' and t belong to the same object (otherwise p could not make C invalid). In order to execute p, the task t' must have obtained the object's lock in a previous step $St_j \rightarrow^{p'} St_{j+1}$ ($j < i$, $ob(o, a, \bot) \in St_j$ and $ob(o, a, t') \in St_{j+1}$) of tr^{lv}. We denote $trace_{p\rightarrow p'}$ the subtrace of tr^{lv} from St_j to St_i. According to the semantics of the language (see Fig. 1) a lock can only be obtained at the beginning of execution of the task t' or at a release point (**await** y?).

[5] At this point we assume a specific shape of tr^{no-rst} that starts and ends with a tr^{w} subtrace. However, the same reasoning can be applied assuming tr^{no-rst} starts or ends with tr^{inc}.

Let S be a loop scope where p appears $p \sqsubset S$. If there is no release point $p' \sqsubset S$, there must be a step $St_j \rightarrow^{p''} St_{j+1}$ that goes inside the loop scope S. We know that p'' must be executed in tr^{lv} and NVISITS(p'') is smaller than NVISITS(p) because it does not have to count the iterations of S or other loop scopes inside S.

N_ATOMIC_NVISITS(p) computes the number of visits to p'' where p'' is the entry point of the biggest loop scope S such that $p \sqsubset S$ and **await** y? $\not\sqsubset S$ or simply the number of visits to p if there is not such a loop. Therefore, we have that $\sum_{\{p|p\in I_S\}}$ N_ATOMIC_NVISITS(p) is a valid and more precise upper bound on the number of iteration traces that are not well-behaved $rst \leq \sum_{\{p|p\in I_S\}}$ N_ATOMIC_NVISITS(p) and can be applied to the definition of LBOUND.

Finally, it is worth mentioning that N_ATOMIC_NVISITS(p) cannot be used to compute inc because for inc we have to count the total amount a field can be modified regardless of whether it happens in one or several t-interleaving traces.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Albert, E., Arenas, P., Correas, J., Genaim, S., Gómez-Zamalloa, M., Román-Díez, G.P., Puebla, G.: Object-sensitive cost analysis for concurrent objects. Softw. Test. Verif. Reliab. **25**(3), 218–271 (2015). doi:10.1002/stvr.1569
3. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: Static analyzer for concurrent objects. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems—20th International Conference, TACAS 2014. Lecture Notes in Computer Science, vol. 8413, pp. 562–567. Springer (2014). doi:10.1007/978-3-642-54862-8_46
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost analysis of concurrent OO programs. In: Yang, H. (ed.) Programming Languages and Systems-9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5–7, 2011. Proceedings, Lecture Notes in Computer Science, vol. 7078, pp. 238–254. Springer (2011). doi:10.1007/978-3-642-25318-8_19
5. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. J. Autom. Reason. **46**(2), 161–203 (2011). doi:10.1007/s10817-010-9174-1
6. Albert, E., Arenas, P., Genaim, S., Puebla, G.: A practical comparator of cost functions and its applications. Sci. Comput. Progr. **111**(3), 483–504 (2015). doi:10.1016/j.scico.2014.12.001
7. Albert, E., Correas, J., Johnsen, E.B., Román-Díez, G.: Parallel cost analysis of distributed systems. In: Static Analysis-22nd International Symposium, SAS 2015. Proceedings, Lecture Notes in Computer Science, vol. 9291, pp. 275–292. Springer (2015). doi:10.1007/978-3-662-48288-9_16
8. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Quantified abstract configurations of distributed systems. Form. Asp. Comput. **27**(4), 665–699 (2015). doi:10.1007/s00165-014-0321-z
9. Albert, E., Correas, J., Román-Díez, G.: Non-cumulative resource analysis. In: Proceedings of 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). Lecture Notes in Computer Science, vol. 9035, pp. 85–100. Springer (2015). doi:10.1007/978-3-662-46681-0_6
10. Albert, E., Flores-Montoya, A., Genaim, S.: Analysis of may-happen-in-parallel in concurrent objects. In: Giese, H., Rosu, G. (eds.) Formal Techniques for Distributed Systems-Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13–16, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7273, pp. 35–51. Springer (2012). doi:10.1007/978-3-642-30793-5_3
11. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: Termination and cost analysis of loops with concurrent interleavings. In: Hung, D.V., Ogawa, M. (eds.) Automated Technology for Verification and Analysis-11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15–18, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8172, pp. 349–364. Springer (2013). doi:10.1007/978-3-319-02444-8_25
12. Albert, E., Genaim, S., Gordillo, P.: May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization. In: Static Analysis-22nd International Symposium, SAS 2015. Pro-

ceedings, Lecture Notes in Computer Science, vol. 9291, pp. 72–89. Springer (2015). doi:10.1007/978-3-662-48288-9_5
13. Albert, E., Gómez-Zamalloa, M., Isabel, M.: Combining static analysis and testing for deadlock detection. In: Integrated Formal Methods-12th International Conference, IFM 2016, Reykjavik, Iceland, June 1–5, 2016. Proceedings, Lecture Notes in Computer Science, vol. 9681, pp. 409–424. Springer (2016)
14. Albert, E., Gómez-Zamalloa, M., Isabel, M.: Syco: A systematic testing tool for concurrent objects. In: Zaks, A., Hermenegildo, M.V. (eds.) Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12–18 2016, pp. 269–270. ACM (2016)
15. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Proceedings of the SAS'10, LNCS, vol. 6337, pp. 117–133. Springer (2010)
16. Armstrong, J., Virding, R., Wistrom, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall, Upper Saddle River (1996)
17. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating runtime and size complexity analysis of integer programs. In: Ábrahám, E., Havelund, K. (eds.) 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014). Lecture Notes in Computer Science, vol. 8413, pp. 140–155. Springer (2014)
18. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 467–478. ACM, New York (2015). doi:10.1145/2737924.2737955
19. Cook, B., Podelski, A., Rybalchenko, A.: Proving thread termination. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pp. 320–330. ACM, New York (2007). doi:10.1145/1250734.1250771
20. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)
21. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: de Nicola, R. (ed.) Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24–April 1, 2007. Proceedings, Lecture Notes in Computer Science, vol. 4421, pp. 316–330. Springer (2007)
22. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: ESOP, Lecture Notes in Computer Science, vol. 2305, pp. 262–277. Springer (2002)
23. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Programming Languages and Systems-12th Asian Symposium, APLAS 2014, Singapore, November 17–19, 2014. Proceedings, LNCS, vol. 8858, pp. 275–295. Springer (2014)
24. Garcia, A., Laneve, C., Lienhardt, M.: Static analysis of cloud elasticity. In: Falaschi, M., Albert, E. (eds.) Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14–16, 2015, pp. 125–136. ACM (2015). doi:10.1145/2790449.2790524
25. Gotsman, A., Cook, B., Parkinson, M.J., Vafeiadis, V.: Proving that non-blocking algorithms don't block. In: Shao, Z., Pierce, B.C. (eds.) Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 16–28. ACM (2009). doi:10.1145/1480881.1480886
26. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2–3), 202–220 (2009). doi:10.1016/j.tcs.2008.09.019
27. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.C., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects-9th International Symposium, FMCO 2010, Graz, Austria, November 29–December 1, 2010. Revised Papers, Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2012)
28. Kupriyanov, A., Finkbeiner, B.: Causal termination of multi-threaded programs. In: Biere, A., Bloem, R. (eds.) 26th International Conference on Computer Aided Verification (CAV 2014). Lecture Notes in Computer Science, vol. 8559, pp. 814–830. Springer (2014)
29. Popeea, C., Rybalchenko, A.: Compositional termination proofs for multi-threaded programs. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'12, pp. 237–251. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_17
30. Schäfer, J., Poetzsch-Heffter, A.: JCobox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010-Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21–25, 2010. Proceedings, LNCS, vol. 6183, pp. 275–299. Springer (2010)
31. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Proceeding of Computer Aided Verification 2014, vol. 8559, pp. 745–761. Springer (2014)

32. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: an adequate abstraction for complexity analysis of imperative programs. CoRR abs/1508.04958 (2015). http://arxiv.org/abs/1508.04958
33. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In: Vitek, J. (ed.) ECOOP 2008-Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7–11, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5142, pp. 104–128. Springer (2008)
34. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: Yahav, E. (ed.) SAS, Lecture Notes in Computer Science, vol. 6887, pp. 280–297. Springer (2011)