Termination and Cost Analysis of Loops with Concurrent Interleavings [⋆]

Elvira Albert[1], Antonio Flores-Montoya[2], Samir Genaim[1], and Enrique Martin-Martin[1]

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

Abstract. By following a *rely-guarantee* style of reasoning, we present a novel termination analysis for concurrent programs that, in order to prove termination of a considered loop, makes the assumption that the "shared-data that is involved in the termination proof of the loop is modified a finite number of times". In a subsequent step, it proves that this assumption holds in all code whose execution might interleave with such loop. 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 with the considered loop. Interestingly, the same kind of reasoning can be applied to 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.

1 Introduction

We develop new techniques for cost and termination analyses of *concurrent objects*. 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) [18,19] 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* [11]. The underlying concurrency model of actor languages forms the basis of the programming languages Erlang [7] and Scala [14] that have gained in popularity, in part due to their support for scalable concurrency. There are also implementations of actor libraries for Java.

[⋆] This work was funded partially by the projects FP7-ICT-610582, TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465.

Termination analysis of concurrent and distributed systems is receiving considerable attention [17,2,9]. 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 [12] and analysis [9] 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 to prove is the *finiteness* of the shared-data involved in the termination proof, i.e., proving that such shared-memory is updated a finite number of times. Our method is based on a circular style of reasoning since the finiteness assumptions are proved by proving termination of the loops in which that shared-memory is modified. Crucial for accuracy is the use of the information inferred by a *may-happen-in-parallel* (MHP) analysis [4], 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. To the best of our knowledge this is the first approach to infer the cost of loops with concurrent interleavings.

Our analysis has been implemented, and its termination component is already fully integrated in COSTABS [2], a COSt and Termination 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.

The rest of the paper is organized as follows. Sec. 2 contains preliminaries about the language, termination and cost. Sec. 3 and 4 explains the rely-guarantee termination and cost analysis, respectively. Sec. 5 contains the preliminary evaluation of the analyses. Finally, Sec. 6 presents the conclusions and related work.

2 Concurrency Model, Termination and Cost

This section presents the syntax and concurrency model of the concurrent objects language, which is basically the same as [15,2]. A *program* consists of a set of classes, each of them can define a set of fields, and a set of methods. The notation $\bar{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. The set of types includes the classes and the set of *future* variable types $\mathit{fut}(T)$. *Pure* expressions pu (i.e., functional expressions that do not access the shared memory) and primitive types are standard and omitted. The abstract syntax of class declarations CL, method declarations M, types T, variables V, and statements s is:

CL ::=**class** $C\ \{\bar{T}\ \bar{f}; \bar{M}\}$ $\quad M$ ::=$T\ m(\bar{T}\ \bar{x})\{s;$**return** $p;\}$ $\quad V$ ::=$x \,|\,$ **this**.f
s ::=$s; s \mid x = e \mid V = x \mid$ **await** $V? \mid$ **if** p **then** s **else** $s \mid$ **while** p **do** s
e ::=**new** $C(\bar{V}) \mid V!m(\bar{V}) \mid pu$ $\quad T$ ::=$C \,|\, \mathit{fut}(T)$

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. Fields are always accessed using the **this** object, and any other object can only access such fields through method calls. We assume that every method ends with a **return** instruction. The concurrency model is as follows. Each object has a lock that is shared by all tasks that belong to the object. Data synchronization is by means of future variables: An **await** y? instruction is used to synchronize with the result of executing task y=$x!m(\bar{z})$ such that **await** y? is executed only when the future variable y is available (i.e., the task is finished). In the meantime, the object's lock can be released and some other *pending* task on that object can take it. W.l.o.g, we assume that all methods in a program have different names.

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 the 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 $\bot$ 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 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 or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created objects and tasks never disappear from the state. Complete semantic rules can be found in the extended version of this paper [5].

Example 1. Figure 1 shows some simple examples which will illustrate different aspects of our analysis. We have an interface Task, and a 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

```
 1 Class TaskQueue{
 2 List<Task> pending=Nil;
 3 void AddTask(Task tk){
 4  pending= appendright(pending,tk);
 5 }
 6 void AddTasks(List<Task> list){
 7  while (list != Nil) {
 8    Task tk = head(list);
 9    pending = tail(list);
10    Fut f=this!AddTask(tk);
11    await f?;}
12 }
13 void ConsumeAsync(){
14  while (pending != Nil) {
15    Task tk = head(pending);
16    pending = tail(pending);
17    Fut f=tk!start();}
18 }
19 void ConsumeSync(){
20  while (pending != Nil) {
21    Task tk = head(pending);
22    pending = tail(pending);
23    Fut f=tk!start();
24    await f?;}
25 }} //end class TaskQueue
26 Interface Task {void start();}
```

```
27 //implementations of main methods
28 main1(List<Task> l){
29   TaskQueue q=new TaskQueue();
30   q!AddTasks(l);
31   q!ConsumeAsync();
32 }
33 main2(List<Task> l){
34   TaskQueue q= new TaskQueue();
35   Fut f=q!AddTasks(l);
36   await f?;
37   q!ConsumeSync();
38 }
39 main3(List<Task> l){
40   TaskQueue q= new TaskQueue();
41   q!AddTasks(l);
42   q!ConsumeSync();
43 }
44 main4(List<Task> l){
45   TaskQueue q= new TaskQueue();
46    while (true){
47       Fut x=q!AddTasks(l);
48       Fut y=q!ConsumeSync();
49       await x?;
50       await y?;}
51 }
```

Fig. 1. Simple examples for termination and cost

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 and tail to get the remaining elements. 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. The former method starts all tasks (L17) concurrently. Instead, method ConsumeSync executes each task synchronously. It releases the processor and waits until the task is finished at L24. In the right-most column, there are four implementations of main methods which are defined outside the classes. Here we show some execution steps from main3:

$St_1 \equiv \{obj(0, f, 0)\ tsk(0, \mathsf{main3}, 0, l, \mathsf{q{=}new\ TaskQueue();...})\} \xrightarrow{new}$

$St_2 \equiv \{obj(0, f, 0)\ obj(1, f_1, \bot)\ tsk(0, \mathsf{main3}, 0, l', \mathsf{q!AddTasks(l);...})\} \xrightarrow{async-call}$

$St_3 \equiv \{obj(0, f, 0)\ obj(1, f_1, \bot)\ tsk(0, \mathsf{main3}, 0, l', \mathsf{q!ConsumeSync(1);...})$

$\quad tsk(1, \mathsf{AddTasks}, 1, l'', \mathsf{while(list!{=}\ Nil);...})\} \xrightarrow{async-call}$

$St_4 \equiv \{obj(0, f, 0)\ obj(1, f_1, \bot)\ tsk(0, \mathsf{main3}, 0, l', \mathsf{return;})\ tsk(1, \mathsf{AddTasks}, 1, l'', ...)$

$\quad tsk(2, \mathsf{ConsumeSync}, 1, l''', \mathsf{while(pending!{=}\ Nil);...})\} \xrightarrow{return} \xrightarrow{activate}$

$St_5 \equiv \{obj(0, f, \bot)\ obj(1, f_1, 2)\ tsk(0..)\ tsk(1..)\ tsk(2..)\}$

Observe that the execution of **new** at St_1 creates the object identified by 1. Then, the executions of the asynchronous calls at St_2 and St_3 spawn new tasks on object `1` identified by `1` and `2`, respectively. In St_4, we perform two steps, first the execution of task `0` terminates (executes return) and object `0` becomes idle, next object `1` (which was idle) selects task `2` for execution. Note that as scheduling is non-deterministic any of both pending tasks (`1` or `2`) could have been selected.

2.1 Termination and Cost

Traces take the form $t \equiv St_0 \to^{b_0} \cdots \to^{b_{n-1}} St_n$, where St_0 is an initial state in which only the `main` method is available and the superscript b_i is the instruction that is executed in the step. A trace is *complete* if it cannot continue from St_n(not taking into account spurious cycles of take-release an object's lock). A trace is *finished* if every task in the configuration $tsk(t, m, o, l, s) \in \mathtt{T}$ is finished $s = \epsilon(v)$). If a trace is complete but not finished, the trace must be *deadlocked*. 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. As we have seen, since we have no assumptions on scheduling, from a given state there may be several possible *non-deterministic* execution steps that can be taken. We say that a program is *terminating* if all possible traces from the initial state are complete.

When measuring the cost, different metrics can be considered. A cost model is a function $\mathcal{M} : \mathit{Ins} \mapsto \mathbb{R}^+$ which maps instructions built using the grammar above to positive real numbers and, in this way, it defines the considered metrics. The cost of an execution step is defined as $\mathcal{M}(St \to^b St') = \mathcal{M}(b)$, i.e., the cost of the instruction applied in the step. 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 from the initial state. 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 larger than or equal to 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 of the grammar. A cost model that counts the number of visits to a method m is defined as $\mathcal{M}_{visits\_m}(b) = 1$ if $b = x!m(\bar{z})$ and 0 otherwise. Consider the partial trace of Ex. 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.

3 Termination Analysis

This section gives first in Sec. 3.1 the intuition behind our method, then it presents the termination algorithm in Sec. 3.2, and finally it provides the results that we need for its application in cost analysis in Sec. 3.3.

3.1 Basic Reasoning

Our starting point is an analysis [2] that infers the termination (and resource consumption) of concurrent programs by losing all information on the shared-memory at "processor release points" (i.e., at the points in which the processor can switch the execution to another task because of an **await** instruction or a method return). Alternatively, instead of losing all information, it can also use monitor invariants (provided by the user) to force some assumptions on the shared-memory. In the latter alternative, the correctness of the analysis depends on the correctness of the provided invariants (the analysis does not infer nor prove them correct). Let us show the kind of problems that [2] can and cannot solve. Consider the first three implementations of main methods:

– main1 creates a TaskQueue q, adds the list of tasks received as input parameter to it, and executes ConsumeAsync. It is not guaranteed that the tasks are added to the queue when ConsumeAsync starts to execute because, as the call at L30 is not synchronized, the processor can be released at L11 and the call at L31 can start to execute. This is not a problem for termination, since ConsumeAsync is executed without releasing the processor. Hence, the method of [2] can prove all methods terminating.
– in main2 the addition of tasks (i.e., the call to AddTasks at L35) is guaranteed to be terminated when ConsumeSync starts to execute due to the use of **await** at L36. However, the difficulty is that ConsumeSync contains a release point. The method of [2] fails to prove termination because at this release point pending is lost. The key is to detect that there are no concurrent interleavings at L24 in this loop by means of an auxiliary MHP analysis.
– main3 has a loop with concurrent interleavings since `ConsumeSync` is called without waiting for completion of `AddTasks`. Thus, some tasks can be added to the list of pending tasks in the middle of the execution of `ConsumeSync`, resulting in a different ordering in which tasks are executed, or even can be added when `ConsumeSync` has finished and hence start will not be executed at all on them. Proving termination requires developing novel techniques.

Our reasoning is at the level of the strongly connected components (SCCs), denoted $\langle S_1, \ldots, S_n \rangle$, in which the code to be analyzed is split. For each method m, we have an SCC named S_m and for each loop (in the methods) starting at Lx we have an SCC named S_x. The analysis starting from main2 must consider the SCCs: $\langle S_{\mathtt{main2}}, S_{\mathtt{AddTasks}}, S_7, S_{\mathtt{AddTask}}, S_{\mathtt{ConsumeSync}}, S_{20} \rangle$. For simplifying the presentation, we assume that each recursive SCC has a single cut-point (in the corresponding CFG). Moreover, the cut-point is assumed to be the entry of the SCC. In such case, an SCC can be viewed as a simple while loop (i.e., without nested loops) with several possible paths in its body. Nested loops can be transformed into this form, by viewing the inner loops as separate procedures that are called from the outer ones. This, however, cannot be done for complex mutual recursions which are rare in our context. The purpose of this assumption is to simplify the way we count the number of visits to a given program point in Sec. 4.

In order to use the techniques of [2] as a black-box, in what follows, we assume that `seq_termin`(S, F) is a basic termination analysis procedure that receives an SCC S and a set of fields F, and works as follows: (1) given a function *fields* that returns the set of fields accessed in the given scope, for any $f \in \mathit{fields}(S) \setminus F$, it adds the instruction $f = *$ at each release point of S; (2) it tries to proves termination of the instrumented code using an off-the-shelf termination analyzer for sequential code; and (3) it returns the result. We assume that `seq_termin` ignores calls to SCCs transitively invoked from the considered scope S, assumes nothing about their return values, and ignores the instruction **await**.

Algorithm 1 MHP-based Termination Analysis

```
1: function TERMINATES($S$,$SSet$)
2:   if $S \in SSet$ then return false
3:   if seq_termin($S, \emptyset$) then return true
4:   $F$ = select_fields($S$)
5:   if (not seq_termin($S, F$)) then return false
6:   $RP$ = release_points($S$)
7:   $MP$ = MHP_pairs($RP$)
8:   $I$ = field_updates($MP, F$)
9:   $DepSet$ = extract_sccs($I$)
10:  for each $S' \in DepSet$ do
11:    if (not TERMINATES($S', SSet \cup \{S\}$) then return false
12:  return true
```

Observation 1 (finiteness assumption) *If S terminates under the assumption that a set of fields F are not modified at the release points of S, then S also terminates if they are modified a finite number of times.*

The intuition behind our observation is as follows. Since the fields are modified finitely, then we will eventually reach a state from which that state on they are not modified. From that state, we cannot have non-termination since we know that S terminates if the fields are not modified. Moreover, one can construct a lexicographical ranking function [8] that witnesses the termination of S.

Example 3. Consider the following two loops:

S_1
```
52 while ( f > 0 ) {
53   x = g();
54   await x?;
55   f−−; }
56
```

S_2
```
57 while ( m > 0 ) {
58   x = g();
59   await x?;
60   f=∗;
61   m−−; }
```

and assume that S_1 and S_2 are the only running processes. Their execution might interleave since both loops have a release point. We let f be a shared variable, m a local variable, and we ignore the behavior of method g. It is easy to see that (a) S_1 terminates under the assumption that f does not change at the release point (L54), and that $RF_1(m, f) = f$ is a ranking function that witnesses its termination; and (b) S_2 terminates without any assumption and $RF_2(m, f) = m$ is a ranking function that witnesses its termination. Since S_2 terminates, we know that f is modified a finite number of times at the release point of S_1 and thus, according to Observation 1, S_1 terminates when running in parallel with S_2. The lexicographical ranking function $RF_3(f, m) = \langle m, f\rangle$ is a witness of the termination of S_1.

3.2 Termination Algorithm

Algorithm 1 presents the main components of our termination algorithm, defined by means of function TERMINATES. The first parameter S is an SCC that we want to prove terminating, and the second one $SSet$ includes the SCCs whose termination requires the termination of S. The role of the second parameter is to detect circular dependencies. In order to prove that a program P terminates, we prove that all its SCCs terminate by calling $\textsc{terminates}(S, \emptyset)$ on each one of them. Let us explain the different lines of the algorithm:

1. At Line 2, if S is in the set $SSet$, then a circular dependency has been detected, i.e., the termination of S depends on the termination of S itself. In such case the algorithm returns **false** (since we cannot handle such cases).
2. At Line 3, it first tries to prove termination of S without any assumption on the fields, i.e., assuming that their values are lost at release points. If it succeeds, then it returns **true**. Otherwise, in the next lines it will try to prove termination w.r.t. some finiteness assumptions on the fields.
3. At Line 4, it selects a set of fields F and, at Line 5, it tries to prove that S terminates when assuming that fields from F keep their values at the release points. If it fails, then it returns **false**. Otherwise, in the next lines it will try to prove that these fields are modified finitely in order to apply Observation 1. The simplest strategy for constructing F (which is the one implemented in our system) is to include all fields used in S. This can also be refined to select only those that might affect the termination of S (using some dependency analysis or heuristics).
4. At this point the algorithm identifies all instructions that might modify a field from F while S is waiting at a release point. This is done as follows: at Line 6 it constructs the set RP of all release points in S; at Line 7 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 [4]); and at Line 8 it remains with $I \subseteq MP$ that actually update a field in F.
5. At Line 9, it constructs a set $DepSet$ of *all* SCCs that can reach a program point in I, i.e., those SCCs that include a program point from I or can reach one by (transitively) calling a method that includes one. Proving termination of these SCCs guarantees that each instruction in I is executed finitely, and thus the fields in F are updated finitely and the finiteness assumption holds.
6. The loop at Line 10 tries to prove that each SCC in $DepSet$ terminates. If it finds one that might not terminate, it returns **false**. In the recursive call S is added to the second parameter in order to detect circular dependencies.
7. If the algorithm reaches Line 12, then S is terminating and returns **true**.

Essentially our approach translates the concurrent program into a sequential setting using the assumptions. To define our proposal, we have focused exclusively on the finiteness assumption because of its wide applicability for proving termination of different forms of loops. Being more general requires a more complex reasoning than when handling other kinds of simpler assumptions. For instance, simpler assumptions (like checking that a field always increases or decreases its value when it is updated) can be easily handled by adding a corresponding test, after Line 8, that checks the assumption holds on the instructions in I.

Example 4. We can now prove termination of both main2 and main3. For main2, the challenge is to prove termination of `ConsumeSync` and namely of the loop that forms S_{20}. This loop depends on the field pending whose size is decreased at each iteration. However, there is a release point in the loop's body (L24). Thus, we need to guarantee the finiteness assumption on pending at that point. The MHP analysis infers that the only other instruction that updates pending at L4 cannot happen in parallel with the release point. This can be inferred thanks to the use of **await** at L11 and L36. Therefore, the set I at Line 8 of Alg. 1 is empty and TERMINATES returns **true**. In the analysis of main3, when proving termination of $S_{\mathsf{ConsumeSync}}$ we have that L4 can happen in parallel with L24 so we have to prove the *finiteness assumption* recursively. In particular, $DepSet = \{S_{\mathtt{AddTask}}, S_7, S_{\mathtt{AddTasks}}, S_{\mathtt{main3}}\}$. Proving termination of S_7 is done directly by `seq_termin` as termination of the loop depends only on the non-shared data list. Also, $S_{\mathtt{AddTask}}$, $S_{\mathtt{AddTasks}}$ and $S_{\mathtt{main3}}$ are proved terminating by `seq_termin` as they do not contain loops. Thus, pending can only increase up to a certain limit and the termination of $S_{\mathsf{ConsumeSync}}$ and all other scopes can be guaranteed.

We can achieve further precision by replacing `extract_sccs` by a procedure `extract_mhp_sccs` which returns all SCCs that can reach a program point in I *and* that can happen in parallel with a release point in RP. A sufficient condition for an SCC to happen in parallel with a point in RP is that its entry point (entry point of while rule) might happen in parallel with a point in RP. The correctness of this enhancement is proved in [5]. The point is that with `extract_sccs` we could find loops that contain I but cannot iterate at RP. These do not have to be taken into account because during the execution of S they will be stopped in a single iteration and therefore cannot cause unboundedness in S. This happens in the next example.

Example 5. Using `extract_mhp_sccs` we can prove that ConsumeSync always terminates in the context of main4. This is true because only one instance of AddTasks is running in parallel with ConsumeSync (due to the **await**s at L49 and L50), and AddTasks is terminating. Using `extract_sccs`, we would detect that L4 is reached from S_{46} and thus, it cannot be proved bounded (due to the **while** (**true**)). However, the MHP analysis tells us that the **await** in L24 of ConsumeSync can run in parallel with AddTasks but not with S_{46}. This reduces the number of SCC we have to consider (removing S_{46}) and thus we can prove ConsumeSync terminating.

Proving termination of the SCCs given by `extract_mhp_sccs` guarantees that each instruction in I is executed finitely during the release points RP, and thus the fields in F are updated finitely and the finiteness assumption holds. We assume that `extract_mhp_sccs` is used in what follows. The following theorem ensures the soundness of our approach (the proof is in [5]).

Theorem 1 (soundness). *Given a program P and its set of recursive SCCs $SSet$. If, $\forall S \in SSet$,* TERMINATES$(S, \emptyset)$ *returns **true**, then P is terminating.*

3.3 Inferring Field-Boundedness

The termination procedure in Sec. 3 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. The *upper* (resp. *lower*) bound of a field f is denoted as f^+ (resp. f^-), and we use f^b to refer to the bounds $[f^-, f^+]$ for f.

Corollary 1. *Consider a field f. If all recursive SCCs that reach a point in which f is updated are terminating, then f is bounded.*

4 Cost Analysis

As for termination, the resource consumption (or cost) of executing a fragment of code can be affected by concurrent interleavings in the loops. Previous work [2] is not able to estimate the cost in these cases. This section proposes new techniques to bound the number of iterations of such loops and thus the cost. This requires to have first proved field-boundedness (Sec. 3.3).

4.1 Cost Analysis of Sequential Programs

Let us first provide an intuitive view of the process of inferring the cost of a program divided in SCCs $S_1, \ldots, S_n$. As an example consider this code:

```
main (int n, int m)
    { int i=0; while (i<n) { i++; s2; int j=i; while (j<m) {s1; j++; }}}
```

where s_1 and s_2 represent a sequence of instructions that do not call any other SCC and do not modify the counters. This leads to one SCC for the inner loop S_1 and one SCC for the outer loop S_2. We first consider the SCC which does not call any other scope, S_1. Given a fragment of sequential code s, we can apply the cost model $\mathcal{M}$ to all instructions in s (see Sec. 2.1) and sum the result, denoted as $\mathcal{M}(s)$. Now, an upper bound on the cost of executing the SCC S_1 is $\texttt{UB}_{S_1} = \#iter{*}\mathcal{M}(body(S_1))$ where $\#iter$ is an upper bound on the number of loop iterations. For sequential programs [3], a ranking function for the loop soundly approximates $\#iter$ and can be automatically inferred. In this case, $\texttt{UB}_{S_1} = \texttt{nat}(m{-}j{+}1){*}\mathcal{M}(body(S_1))$, where function $\texttt{nat}$ is defined as $\texttt{nat}(n) = n$ if $n \geq 0$ and 0 otherwise (it is used to avoid having negative costs [3]).

We consider now the general case in which we need to *compose* the cost of different SCCs. The point is that in order to plug the cost that we have already computed for S_1 in its calling SCC S_2, we need to *maximize* it (i.e., compute its worst case cost). Intuitively, the worst case cost is when j is 0 and hence $\texttt{UB}_{S_1}$ becomes $\texttt{nat}(m{+}1){*}\mathcal{M}(body(S_1))$. Intuitively, maximization works by first inferring an *invariant* that holds between the arguments at the initial call (main method) and at each iteration during the execution. For instance, we infer the invariant $0 \leq j \leq m_0$ which holds in S_1 where m_0 is the initial value for m. Maximizing $\texttt{UB}_{S_1}$ using the invariant results in $\texttt{nat}(m{+}1){*}\mathcal{M}(body(S_1))$. In what follows, we refer as $\texttt{max\_init}(e)$ to the maximization of an expression e using such procedure (see [3]) which we simply adopt in this paper. Thus, the upper bound for S_2 is $\texttt{UB}_{S_2} = \#iter{*}(\mathcal{M}(body(S_2)) + \texttt{max\_init}(\texttt{UB}_{S_1})) \equiv \texttt{nat}(n){*}(\mathcal{M}(body(S_2)){+}\texttt{nat}(m{+}1){*}\mathcal{M}(body(S_1)))$.

Note that if the considered SCC is not recursive, then we simply apply $\mathcal{M}$ to the sequential instructions and compose the SCCs as above. SCCs with multiple recursive calls (that lead to an exponential complexity) and loops with logarithmic complexity are treated analogously, see [3].

4.2 Basic Reasoning

In order to explain the intuition of our approach, let us first consider the sequential loop in S_1 whose termination behavior has been widely studied by the termination community (we use $*$ to ignore irrelevant code):

S_1
```
64 while (f>0){
65  f−−;
66  if (∗ & m>0)
67       { m−−;
68          f=∗;
69 }}
```
S_2
```
70 while (f>0){
71    f−−;
72    await ∗?
73 }
```
S_3
```
74 while (m>0){
75    m−−;
76    f=∗;
77 }
```
Our method is inspired by the observation that, provided the **if** statement is executed a finite number of times, an upper bound on the number of iterations of S_1 can be computed as: the maximum number of iterations of the loop ignoring the **if** statement, but assuming that such **if** statement updates the field f with its maximum value, *multiplied* by the maximum number of times that the **if** statement can be executed. Intuitively, we assume that every time the **if** statement is executed the field can be put to its maximum value and thus the loop can be executed the maximum number of times in the next iteration. Hence, $\texttt{max\_init}(f){*}m$ is an upper bound for the loop, and $\texttt{max\_init}(f) = f^+$ results in the maximum value for field f (see Sec. 3.3).

We propose to apply a similar reasoning to bound the number of iterations of loops with concurrent interleavings. Assume that S_2 and S_3 are the only running processes and that the execution of the instruction at L76 that updates the field may interleave with the **await** in S_2. We have a similar behavior to the leftmost loop, though they are obviously not equivalent. Instead of having an interleaving **if**, we have an interleaving process that updates the field. Our proposal is to first bound the number of times that instruction 76 can be executed. A sound and precise bound is m. Our main observation is that, the upper bound for S_2 is the maximum number of iterations ignoring the **await**, but assuming that at this point f can take its maximum value f^+, multiplied by the maximum number of *visits* to 76. Thus, $f^+{*}m$ is a sound upper bound. If we have a loop like while (f<0) {f++; await ∗?}, whose ranking function is $-f$, then the worst case cost occurs when f is set to its minimum value f^-, i.e., $\texttt{max\_init}(-f) = f^-$. Therefore, maximizing a ranking function that involves a field f is done by relying on its field bound f^b, and it may result, depending on the case, in f^+ or f^-.

Observation 2 (loop bounds) *An upper bound on the number of iterations of a loop l with interleaving instructions that update fields F is* NITER∗(NVISITS*+1):*
1. *where* NVISITS *is the number of visits to the points in which fields in F are updated and that might interleave their execution with the loop release points;*
2. *and* NITER *is the number of iterations of the loop ignoring the interleavings —maximized w.r.t. the bounds for the fields in F;*

Our analysis relies on the assumption that the number of visits (item 1) is bounded, which has been proved in Corollary 1. Given a bound on the number of loop iterations, the cost is obtained as in the sequential case, i.e., by applying the cost model to the instructions in the loop body and multiplying it by our loop bound. Thus, we only focus now on bounding the number of loop iterations.

4.3 Bounding the Number of Iterations for Loops with Interleavings

Alg. 2 presents two mutually recursive functions which allow us to infer the two items of the observation above. For each SCC S, we assume that after executing

Algorithm 2 Bounding the Number of Iterations for Loops with Interleavings

```
1:  function NITER($S, SSet$)
2:    if $S \in SSet$ then return false
3:    if $S$ is not recursive then return 1
4:    $i = 1$;
5:    for each $p \in S_I$ do
6:      $i = i$+NVISITS($p, S_{RP}, SSet \cup S$)
7:    return max_init($S_{RF}$)$*i$

8:  function NVISITS($p, RP, SSet$)
9:    $V_p$= 0;
10:   $P$= mhp_reachable_paths(p,$RP$);
11:   for each $\langle S_1, \ldots, S_n \rangle$ in $P$ do
12:     $V_{aux}$=1;
13:     for $i = 1$ to $n$ do
14:       $V_{aux} = V_{aux}$*NITER($S_i, SSet$)
15:     $V_p = V_p + V_{aux}$
16:   return $V_p$
```

Alg. 1 we have the following information: the set RP computed at Line 6, denoted as S_{RP}; the set I computed at Line 8, denoted as S_I; and a (linear) ranking function computed by the `seq_termin` at Lines 3 and 5, denoted as S_{RF}. If S was proved terminating at Line 3 (i.e., losing the fields), we assume that S_I and S_{RP} are empty. Function NITER receives an SCC S whose number of iterations is to be bounded and a set of SCCs $SSet$ which, as before, is initially empty and allows us to detect cyclic dependencies (Line 2). As the number of SCCs is finite, termination is guaranteed. If the SCC S is not recursive, it simply returns one (Line 3). Otherwise, the number of iterations in the SCC can be bound by the maximization of the local ranking function, multiplied by the maximum number of visits to all the points that update the fields (Line 7) *and* that may happen in parallel with S_{RP} (to this end we pass S_{RP} as parameter to NVISITS). As mentioned in Sec. 4.1, function `max_init` maximizes the received expression w.r.t. the input parameters of the entry method (often main), and the field bounds f^b are used for maximizing the fields.

Function NVISITS receives a program point p, a set of release points RP, and infers an upper bound on the number of visits to p while the program is waiting at a point of RP. We first compute the multiset of reachable paths to p. Each path is of the form $\langle S_1, \ldots, S_n \rangle$, i.e., it is a sequence of SCCs which reach the program point p. For each of the paths (Line 11), we traverse all the SCCs in the path (Line 13) and multiply the number of iterations of the corresponding SCC by those of the SCCs already traversed *if* the SCC might happen in parallel with the release points RP. We assume that `mhp_reachable_paths` gives us only those SCC that may happen in parallel with the release points RP passed as parameters. The number of visits from each of the paths is accumulated to the paths that have been already accounted (Line 15).

Example 6. Let us consider method ConsumeSync invoked from main3. We want to compute NITER($S_{20}, \emptyset$). Alg. 1 gives us that the local ranking function is $RF = length$(pending) and that the program point 4 may happen in parallel with the release point 24 and update the field pending. Hence, we need to compute NVISITS($4, \{24\}, \{S_{20}\}$). We first compute the reachable paths to 4, which gives us the only element $\langle S_{\texttt{AddTask}}, S_7, S_{\texttt{AddTasks}} \rangle$. Note that $S_{\texttt{main3}}$ is not included in the path because its entry point cannot happen in parallel with 24. We start by computing $\text{NITER}(S_{\texttt{AddTask}}, \{S_{20}\})$, since $S_{\texttt{AddTask}}$ is not recursive, we simply return 1 which is multiplied at Line 14 of Alg. 2 by the initial value for V_{aux} (which is 1). The next iteration of the **for** loop at Line 13 invokes $\text{NITER}(S_7, \{S_{20}, S_{\texttt{AddTask}}\})$. In this case, by Alg. 1, we have the local ranking function $length(\mathsf{list})$ and that the set of points at which list is updated is empty. The maximization of $length(\mathsf{list})$ returns it in terms of the initial parameters of main3, i.e., $length(\mathsf{l})$. This value is multiplied at Line 14 by 1 (previous value of V_{aux}). Finally, we compute $\text{NITER}(S_{\texttt{AddTasks}}, \{S_7, S_{20}, S_{\texttt{AddTask}}\})$ that, as it is not recursive, simply returns 1. The execution of the **for** loop at Line 13 finishes and also the execution of the **for** each loop at Line 11 and we have that $\text{NVISITS}(4, \{S_{20}\}) = length(\mathsf{l})$. Thus, we can now finish the computation of $\text{NITER}(S_{20}, \emptyset)$ returning $length(\mathsf{pending}^+) * length(\mathsf{l})$. The upper bound for ConsumeSync when invoked from main4 can be obtained in a similar way.

The following theorem ensures the soundness of our approach. The proof can be found in [5].

Theorem 2 (soundness). *Given a recursive SCC S, the execution of $\text{NITER}(S, \emptyset)$ terminates and returns an upper bound on the number of iterations in S.*

5 Implementation and Preliminary Evaluation

We have implemented the described cost and termination analyses, although currently only the termination component is integrated within COSTABS. Our analysis can be tried online at http://costa.ls.fi.upm.es/costabs by enabling the option "*rely-guarantee termination analysis*". The cost analysis component will be available for its online use from the same site soon. Given a program and a selection of an entry method from which the analysis will start, the output of the analysis is a description of the SCCs (reachable from the entry) which are terminating. This section aims at performing a preliminary experimental evaluation of the accuracy and performance of our implementation, by comparing our results with those obtained by the previous version of the analyzer which loses all information on the shared-memory. For this purpose, we have analyzed a set of small and medium-sized programs, as well as one industrial case study, the *Replication System*, developed by Fredhopper®. The analyzed code for all examples can be found and tried in the above site.

Regarding the small and medium-sized examples, their number of lines of code ranges from 20 to 100 and the number of SCCs from 5 to 20. Both versions of the analyzer need less than 1 sec. to analyze each program. All terminating loops with concurrent interleavings are reported by our rely-guarantee method, improving the results of the previous analyzer. Our largest experiment is performed on the *Replication System*, a case study that provides search and merchandising IT services to e-Commerce companies, developed within the HATS project (http://www.hats-project.eu/). It has 2100 lines of code and 426 SCCs that need to be analyzed. The previous analyzer needs 2813 sec. and proves 420 SCCs terminating, whereas the rely-guarantee method proves 423 SCCs terminating in only 41 sec. Times are obtained as the arithmetic mean of five runs on a Ubuntu 12.04 32-bit with Intel Core2 Quad CPU Q9550 2.83GHz and 3.4GiB of memory. The efficiency of our rely-guarantee method can be explained because it works modularly at the level the SCCs, instead of analyzing the program globally as the previous analyzer. An inspection of the three additional SCCs that have been proved terminating confirms that they indeed correspond to loops with concurrent interleavings. The reason why a simple analysis that loses the shared-memory could achieve already good results is that the (experienced) developers of the case study were aware of the risks of having loops with concurrent interleavings and they were very much avoided.

6 Conclusions and Related 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 [4].

Existing methods for proving termination of thread-based programs also apply a rely-guarantee or assume-guarantee style of reasoning [9,17,10]. 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 [9]. 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 [9].

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 [13,6,20] for inferring complex loop bounds for (sequential) transition systems. 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 that 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!ConsumeSync();. The MHP analysis infers that ConsumeSync can run in parallel with itself. When trying to solve the equations a cyclic dependency is created and both TERMINATES and NITER algorithms terminate returning **false**.

References

1. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *Proc. of APLAS'11*, volume 7078 of *LNCS*, pages 238–254. Springer, December 2011.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
4. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *FORTE'12*, *LNCS* 7273, pages 35–51. Springer, 2012.
5. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings (Extended Version). Technical Report SIC 06/13, Univ. Complutense de Madrid, 2013.
6. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. of SAS'10*, volume 6337 of *LNCS*. Springer, 2010.
7. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1996.
8. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. volume 3576 of *LNCS*, pages 491–504. Springer, 2005.
9. B. Cook, A. Podelski, and A. Rybalchenko. Proving Thread Termination. In *Proc. of PLDI'07*, pages 320–330. ACM, 2007.
10. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.
11. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
12. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-Modular Verification for Shared-Memory Programs. In *ESOP'02*, LNCS 2305, pages 262–277. Springer, 2002.
13. Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010.
14. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
15. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. of FMCO'10 (Revised Papers)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.
16. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Meth.*, 14:1–41, January 2005.
17. C. Popeea and A. Rybalchenko. Compositional Termination Proofs for Multi-Threaded Programs. In *Proc. of TACAS'12*, LNCS 7214. Springer, 2012.
18. J. Schäfer and A. Poetzsch. Jcobox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, LNCS 6183, pages 275–299. Springer, 2010.
19. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008.
20. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, *LNCS* 6887, pages 280–297. Springer, 2011.