May-Happen-in-Parallel Analysis with Returned Futures *

Elvira Albert, Samir Genaim, and Pablo Gordillo

Complutense University of Madrid (UCM), Spain

Abstract. May-Happen-in-Parallel (MHP) is a fundamental analysis to reason about concurrent programs. It infers the pairs of program points that may execute in parallel, or interleave their execution. This information is essential to prove, among other things, absence of data races, deadlock freeness, termination, and resource usage. This paper presents an MHP analysis for asynchronous programs that use *futures* as synchronization mechanism. Future variables are available in most concurrent languages (e.g., in the library `concurrent` of Java, in the standard thread library of C++, and in Scala and Python). The novelty of our analysis is that it is able to infer MHP relations that involve future variables that are *returned* by asynchronous tasks. Futures are returned when a task needs to await for another task created in an *inner* scope, e.g., task t needs to await for the termination of task p that is spawned by task q that is spawned during the execution of t (not necessarily by t). Thus, task p is awaited by task t which is in an outer scope. The challenge for the analysis is to (back)propagate the synchronization of tasks through future variables from inner to outer scopes.

1 Introduction

MHP is an analysis of utmost importance to ensure both liveness and safety properties of concurrent programs. The analysis computes *MHP pairs*, which are pairs of program points whose execution might happen, in an (concurrent) interleaved way within one processor, or in parallel across different processors. This information is fundamental to prove absence of data races as well as more complex properties: In [13], MHP pairs are used to discard unfeasible deadlock cycles; namely if a deadlock cycle inferred by the deadlock analyzer includes pairs of program points that are proven not to happen in parallel by our MHP analysis, the cycle is spurious and the program is deadlock free. In [4], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. For instance, consider a loop whose termination cannot be proven because of a potential execution in parallel of the loop with a task that modifies the variables that control the loop guard (and thus threatens

* This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, by the CM project S2013/ICE-3006 and by the UCM CT27/16-CT28/16 grant.

its termination). If our MHP analysis proves the unfeasibility of such parallelism, then termination of the loop can be guaranteed.

For simplicity, we develop our analysis on a small asynchronous language which uses *future variables* [12, 10] for task synchronization. A method call m on some parameters $\bar{x}$, written as f=m($\bar{x}$), spawns an asynchronous task, and the future variable f allows synchronizing with the termination of such task by means of the instruction **await** f?; which delays the execution until the asynchronous task has finished. In this fragment of code f=m(..) ;...; **await** f?; the execution of the instructions of the asynchronous task m may happen in parallel with the instructions between the asynchronous call and the **await**. However, due to the future variable in the **await** instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the **await**. Therefore, future variables play an essential role within an MHP analysis and it is essential for its precision to track them accurately. Future variables are available in most concurrent languages: Java, Scala and Python allow creating pools of threads. The users can submit tasks to the pool, which are executed when a thread of the pool is idle, and may return future variables to synchronize with the tasks termination. C++ includes the components async, future and promise in its standard library, which allow programmers to create tasks (instead of threads) and return future variables in the same way as we do.

In this paper, we present to the best of our knowledge the first MHP analysis that captures MHP relations that involve tasks that are awaited in an outer scope from the scope in which they were created. This happens when future variables are returned by the asynchronous tasks, as it can be performed in all programming languages that have future variables. Our analysis builds on top of an existing MHP analysis [3] that was extended to track information of future variables passed through method parameters in [5], but it is not able to track information propagated through future variables that are returned by tasks. The original MHP analysis [3] involves two phases: (1) a local analysis which consists in analyzing the instructions of the individual tasks to detect the tasks that it spawns and awaits, and (2) a global analysis which propagates the local information compositionally. Accurately handling returned future variables requires non-trivial extensions in both phases:

1. The local phase needs to be modified to backpropagate the additional inter-procedural relations that arise from the returned futures variables. Back-propagation is achieved by modifying the data-flow of the analysis so that it iterates to propagate the new dependencies.
2. The global phase has to be modified by reflecting in the analysis graph the additional information provided by the local phase. A main achievement has been to generate the necessary information at the local phase so that the process of inferring the MHP pairs remains as in the original analysis.

Our analysis has been implemented within the SACO static analyzer [2], which is able to infer the safety and liveness properties mentioned above. The system can be used online at `http://costa.ls.fi.upm.es/saco/web/`, where the bench-

marks used in the paper are available. Our experiments show that our analysis improves the accuracy over the previous analysis with basically no overhead.

2 Language

We present the syntax and semantics of the asynchronous language on which we develop our analysis. A program P is composed by a set of classes. Each class contains a set of fields and a set of methods. A (concurrent) object of a class represents a processor with a queue of tasks which (concurrently) execute the class methods, and access a shared-memory made up by the object fields. One of the tasks will be active (executing) and the others pending to be executed. The notation $\overline{M}$ is used to abbreviate $M_1, ..., M_n$. Each field and method has a type T. The set of types includes class identifiers C and future variable types $fut\langle T\rangle$. A method receives a set of variables as arguments $\bar{x}$, contains local variables $\bar{x}'$, a returned variable, and a sequence of instructions s.

$$
\begin{array}{lll}
CL & ::= & class\ C\ \{\overline{T}\ \overline{f}; \overline{M}\} \\
M & ::= & T\ m(\overline{T}\ \bar{x})\ \{\overline{T}\ \bar{x}'; s\} \\
s & ::= & \epsilon \mid b; s \\
b & ::= & o = \textsf{new}\ C(\bar{x}) \mid \textsf{if}\ (*)\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{while}\ (*)\ \textsf{do}\ s \mid y = o.m(\bar{x}) \mid \\
 & & \mid \textsf{await}\ y? \mid z = y.\textsf{get} \mid \textsf{return}\ y \mid \textsf{skip}
\end{array}
$$

y and z represent variables of type $fut\langle T\rangle$ and x represents a variable of type T. Arithmetic expressions are omitted for simplicity and are represented by the instruction **skip**. This instruction has no effect on the analysis of the program. The loop and conditional statements are non-deterministic and the symbol $*$ represents *true* or *false*. The instruction $y = o.m(\bar{x})$ corresponds to an asynchronous call. It spawns a new instance of the task m in the object o and binds the task to the future variable y. Instruction **await** $y?$ is used to synchronize with the task $y = o.m(\bar{x})$, and blocks the execution in object o until task m finishes its execution. $z = y.\textsf{get}$ retrieves the value returned by the method bound to y and associates it with z. W.l.o.g., we make the following assumptions: each **get** instruction is preceded by an **await**, i.e., the task associated to the **get** statement has to be finished to access its returned value; the program has a method call main without parameters from which the execution will start; future variables can be used once and they cannot be reused after they are bound to a task; the **get** instruction can be applied once over each future variable; we restrict the values returned by a method to future variables; each method can only have a **return** statement in its body, and it has to be the last instruction of the sequence. We let $\texttt{ppoints}(m)$ and $\texttt{ppoints}(P)$ be the set of program points of method m and program P respectively, $\texttt{methods}(P)$ be the set of method names of program P and $\texttt{futures}(P)$ be the set of all future variables defined in program P.

Let us define the operational semantics for the language. A *program state* S is a tuple $S = \langle O, T\rangle$ where O is the set of objects and T is the set of tasks. Only one task can be active in each object. An *object* is a term $obj(o, a, lk)$ where o is

$$(1)\ \frac{l' = l[o \to bid_1], O' = O \cup \{obj(bid_1, a, \bot)\}, a = init\_atts(C, \bar{x}), bid_1 \text{ is a fresh id}}{\langle O, \{tsk(tid, m, l, bid, \top, o = \textbf{new } C(\bar{x}); s) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tid, m, l', bid, \top, s) \parallel T\}\rangle}$$

$$(2)\ \frac{l(o) = bid_1 \neq \textbf{null}, l' = l[y \to tid_1], l_1 = buildLocals(\bar{x}, m)), tid_1 \text{is a fresh id}}{\begin{array}{c}\langle O, \{tsk(tid, m, l, bid, \top, y{=}o.m_1(\bar{x}); s) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tid, m, l', bid, \top, s), tsk(tid_1, m_1, l_1, bid_1, \bot, body(m_1)) \parallel T\}\rangle\end{array}}$$

$$(3)\ \frac{l_1(y) = tid_2}{\begin{array}{c}\langle O, \{tsk(tid_1, m_1, l_1, bid_1, \top, \textbf{await } y?; s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tid_1, m_1, l_1, bid_1, \top, s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle\end{array}}$$

$$(4)\ \frac{l_1(y) = tid_2, l_1' = l_1[z \to v]}{\begin{array}{c}\langle O, \{tsk(tid_1, m_1, l_1, bid_1, \top, z{=}y.\textbf{get}; s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{tsk(tid_1, m_1, l_1', bid_1, \top, s_1), tsk(tid_2, m_2, l_2, bid_2, \bot, \epsilon(v)) \parallel T\}\rangle\end{array}}$$

$$(5)\ \frac{obj(bid, a, \top) \in O, O' = O[obj(bid, a, \top)/obj(bid, a, \bot)], v = l(y)}{\langle O, \{tsk(tid, m, l, bid, \top, \textbf{return } y) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tid, m, l, bid, \bot, \epsilon(v)) \parallel T\}\rangle}$$

$$(6)\ \frac{\begin{array}{c}(l', s') = eval(instr, O, l) \\ instr \in \{\textbf{skip}, \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2, \textbf{while } b \textbf{ do } s_3\}\end{array}}{\langle O, \{tsk(tid, m, l, bid, \top, instr; s) \parallel T\}\rangle \rightsquigarrow \langle O, \{tsk(tid, m, l', bid, \top, s') \parallel T\}\rangle}$$

$$(7)\ \frac{obj(bid, a, \bot) \in O, O' = O[obj(bid, a, \bot)/obj(bid, a, \top)], s \neq \epsilon(v)}{\langle O, \{tsk(tid, m, l, bid, \bot, s) \parallel T\}\rangle \rightsquigarrow \langle O', \{tsk(tid, m, l, bid, \top, s) \parallel T\}\rangle}$$

Fig. 1. Summarized Semantics

the identifier of the object, a is a mapping from the object fields to their values and $lk \in \{\top, \bot\}$ indicates whether the object contains an active task executing ($\top$) or not ($\bot$). A task is a term $tsk(t, m, l, o, lk, s)$ where t is a unique task identifier, m is the method name that is being executed, l is a mapping from the variables of the task to their values, o is the identifier of the object in which the task is executing, $lk \in \{\top, \bot\}$ indicates if the task has the object's lock or not and s is a sequence of instructions that the task will execute or $s = \epsilon(v)$ if the task has finished and the return value v is available. The execution of a program starts from the initial state $S_0 = \langle obj(0, a, \top), tsk(0, \mathsf{main}, l, 0, \top, body(main)\rangle$ where a is an empty mapping, and l maps future variables to **null**.

The execution starts from S_0 applying *non-deterministically* the semantic rules from Fig. 1. We use the notation $\{t \parallel T\}$ to represent that task t is the one selected non-deterministically for the execution. At each step, a subset of the state S is rewritten according to the rules of Fig. 1 as follows: (1) creates a new object with an empty queue, free lock and initializes its fields ($init\_atts$). (2) corresponds to an asynchronous call. It gets the identifier of the object which

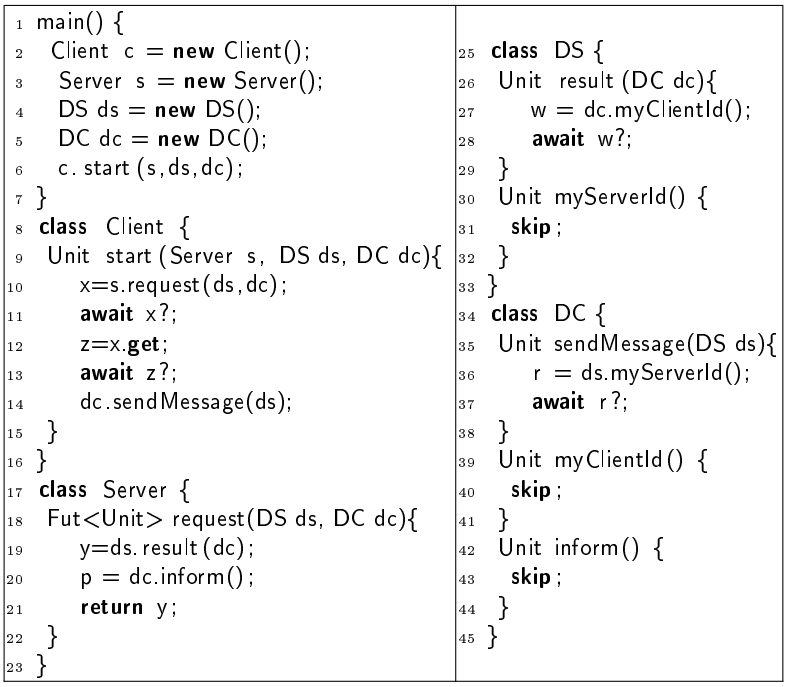
is going to execute the task, initializes the parameters and variables of the task (*buildLocals*), and creates the new task with a new identifier that is associated with the corresponding future variable. (3) An **await** y? statement waits until the task bound to y finishes its execution. (4) checks if the task bound to the future variable involved in the **get** statement is finished. If so, it retrieves the value associated with the future variable. (5) After executing the **return** statement, the retrieved value is stored in v so that it can be obtained by the future variable bound to this task. Then, the object's lock is released ($O[o/o']$ means that the object o is replaced by o' in O) and the task is finished ($\epsilon(v)$ is added to the sequence of instructions). (6) covers sequential instructions that do not affect synchronization by moving the execution of the corresponding task to the next instruction and possibly changing the state (represented by *eval*). Finally, (7) is used to get the object's lock by an unfinished task and start its execution.

In what follows, given a task $tsk(t, m, l, o, lk, s)$, $pp(s)$ denotes the program point of the first instruction of s. If s is empty, $pp(s)$ returns the exit program point of the corresponding method, denoted $exit(m)$. Given a state $S = \langle O, T\rangle$, we define its set of MHP pairs, i.e., the set of program points that can run in parallel as $\mathcal{E}(S) = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, m_1, l_1, o_1, lk_1, s_1), tsk(tid_2, m_2, l_2, o_2, lk_2, s_2) \in T, tid_1 \neq tid_2\}$. The set of MHP pairs for a program P is defined as the set of MHP pairs of all reachable states, namely $\mathcal{E}_P = \cup\{\mathcal{E}(S_n) \mid S_0 \rightsquigarrow^* S_n\}$.

3 Motivation: using MHP pairs in deadlock analysis

Let us motivate our work by showing its application in the context of deadlock analysis. Consider the example in Fig. 2 that models a typical client-server application with two delegate entities to handle the requests. The execution starts from the main block by creating four concurrent objects, the client c, the server s, and their delegates dc and ds, respectively. The call start at Line 6 (L6) spawns an asynchronous task on the client object c that sends as arguments references to the other objects. When this task is scheduled for execution on the client, we can observe that it will spawn an asynchronous task on the server (L10) and another one on the delegate-client (L14). The request task on the server in turn posts two asynchronous tasks on the delegate-server (L19) and delegate-client objects (L20). Such delegates communicate directly with each other as we have passed as arguments the references to them.

The most challenging aspect for the analysis of this model is due to the synchronization through returned future variables. For instance at L12 the instruction x.**get** retrieves the future variable returned by request at L21. Thus, we would like to infer that after L13 the task executing result at the object ds has terminated. The inference needs to backpropagate this synchronization information from the inner scope where the task has been created (L19) to the outer scope where it is awaited (L13). This backpropagation is necessary in order to prove that the execution of this application is deadlock free. Otherwise, an MHP-based deadlock analyzer will spot an unfeasible deadlock. Fig. 3 shows a fragment of the graph that a deadlock analyzer [13] constructs: the concurrent



```
1  main() {
2   Client c = new Client();
3    Server s = new Server();
4    DS ds = new DS();
5    DC dc = new DC();
6    c.start(s,ds,dc);
7  }
8  class Client {
9   Unit start(Server s, DS ds, DC dc){
10      x=s.request(ds,dc);
11      await x?;
12      z=x.get;
13      await z?;
14      dc.sendMessage(ds);
15   }
16 }
17 class Server {
18  Fut<Unit> request(DS ds, DC dc){
19      y=ds.result(dc);
20      p = dc.inform();
21      return y;
22   }
23 }
25 class DS {
26  Unit result(DC dc){
27      w = dc.myClientId();
28      await w?;
29   }
30  Unit myServerId() {
31    skip;
32   }
33 }
34 class DC {
35  Unit sendMessage(DS ds){
36      r = ds.myServerId();
37      await r?;
38   }
39  Unit myClientId() {
40    skip;
41   }
42  Unit inform() {
43    skip;
44   }
45 }
```

Fig. 2. Example of client-server model

objects are in circles, the asynchronous tasks in boxes, and labelled arrows contain the program lines at which tasks post new tasks on the destiny objects. In the bold arrows of the graph, we can observe the cycle detected by the analyzer due to the task result and sendMessage executing respectively in objects ds and dc. These two tasks wait for the termination of tasks myClientId and myServerId in each other object, thus creating a potential cycle. Our MHP analysis will accurately infer that these two tasks cannot happen simultaneously, and will allow the deadlock analyzer to break this unfeasible deadlock cycle. Fig. 4 shows some of the MHP pairs that the analysis in [3] infers, and we mark in bold font those pairs that our analysis spots as spurious (as will be explained along the paper). For instance, the original analysis infers L15||L28 and L37||L28. However, we detect that at program point L14 the task result is finished so it cannot run in parallel with task sendMessage and hence those pairs are eliminated, this allows us later to discard the potential deadlock described above.

4 MHP analysis

The MHP analysis of [3] consists of two phases. The first one, the local phase, considers each method separately and infers information (at each program point of the method) about the status of the tasks that are created locally in that

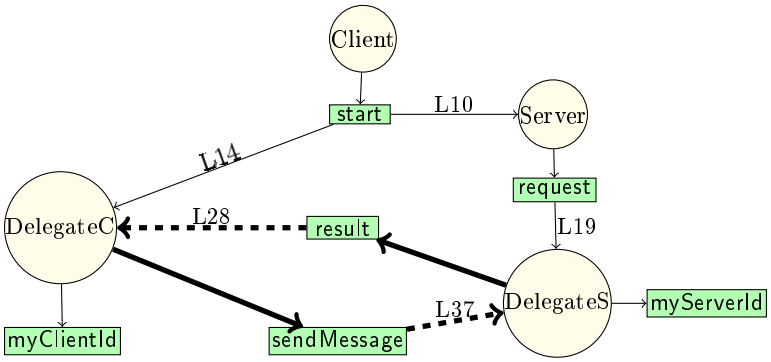


Fig. 3. Partial data-flow graph of example in Fig. 2.

$L11 \| \| L19$	$L11 \| \| L20$	$L11 \| \| L21$	$L11 \| \| L28$	$L11 \| \| L29$	$L11 \| \| L40$	$L12 \| \| L22$
$L12 \| \| L28$	$L12 \| \| L29$	$L12 \| \| L40$	$L14 \| \| L22$	$\mathbf{L14 \| \| L28}$	$L14 \| \| L29$	$\mathbf{L14 \| \| L40}$
$L15 \| \| L22$	$\mathbf{L15 \| \| L28}$	$L15 \| \| L29$	$\mathbf{L15 \| \| L40}$	$L15 \| \| L31$	$L21 \| \| L29$	$L21 \| \| L40$
$L22 \| \| L28$	$L22 \| \| L44$	$L28 \| \| L41$	$L28 \| \| L43$	$L28 \| \| L44$	$L29 \| \| L41$	$L29 \| \| L44$
$L29 \| \| L41$	$L29 \| \| L44$	$\mathbf{L28 \| \| L37}$	$\mathbf{L28 \| \| L38}$	$L29 \| \| L40$	$L21 \| \| L44$	

Fig. 4. Results of MHP analysis.

method. The second one, the global phase, uses the information inferred by the first phase to construct an MHP graph from which an over-approximation of the MHP pairs set can be extracted. As mentioned already, the limitation of this analysis is that it does not track inter-procedural synchronizations originating from (1) passing future variables as method parameters; or (2) returning future variables from one method to another. The work of [5] extends [3] to handle the first issue, and in this paper we extend it to handle the second one. Both extensions require different techniques, and are both complementary and compatible. To simplify the presentation, we have not started from the analysis with future variables as parameters [5], but rather from the original formulation [3]. In Sec. 6, we provide a detailed comparison of [5] and our current extension.

4.1 Local MHP

The local phase of the MHP analysis (LMHP) of [3] considers each method n separately, and for each program point $\ell \in \texttt{ppoints}(n)$ it infers a LMHP state that describes the status of each task invoked in n before reaching ℓ. Formally, a LMHP state E is a *multiset* of MHP atoms, where an MHP atom is:

1. $y{:}\texttt{T}(m, \texttt{act})$, which represents a task that is an instance of method m and can be executing at any program point. We refer to it as *active* task; and

$$
\begin{array}{ll}
(1) & \tau(y = o.m(\bar{x}), E) = E[y{:}\mathtt{T}(m, X)/\star{:}\mathtt{T}(m, X)] \cup \{y{:}\mathtt{T}(m, \mathtt{act})\} \\
(2) & \tau(\textsf{await } y?, E) = E[y{:}\mathtt{T}(m, \mathtt{act})/y{:}\mathtt{T}(m, \mathtt{fin})] \\
(3) & \tau(z = y.\textsf{get}, E) = E' \cup E'' \cup E''' \quad \text{where:}
\end{array}
$$

$$
\boxed{\begin{array}{l}
E' \;= \mathtt{eliminate}(\{y\}, E[z{:}\mathtt{T}(m, X)/\star{:}\mathtt{T}(m, X)]) \\
E'' = \{z{:}\mathtt{T}(n, X) \mid y{:}\mathtt{T}(f, \mathtt{fin}) \in E,\ \mathtt{T}(n, X) \in \mathit{Ret}(f)\} \\
E''' = \{y{:}\mathtt{T}(f, \overline{\mathtt{fin}}) \mid y{:}\mathtt{T}(f, \mathtt{fin}) \in E\}
\end{array}}
$$

$$
(4) \qquad \tau(b, E) = E \qquad \textit{otherwise}
$$

Fig. 5. Local MHP transfer function τ.

2. $y{:}\mathtt{T}(m, \mathtt{fin})$, which represents a task that is an instance of method m and has finished its execution already (i.e., it is at its exit program point). We refer to it as *finished* task.

In both cases, the task is associated to future variable y, i.e., in the concrete state that E describes y is bound to the unique identifier of the corresponding task. Intuitively, the MHP atoms of E represent the tasks that were created locally and are executing in parallel. In what follows, we use $y{:}\mathtt{T}(m, X)$ to refer to an MHP atom without specifying if it corresponds to an active or finished task. MHP atoms might also use the symbol $\star$ instead of a future variable to indicate that we do not know to which future variable, if any, the task is bound. Note that if we have two atoms with the same future variable in a LMHP state E, then they are mutually exclusive, i.e., only one of the corresponding tasks might be executing since at the concrete level y can be bound only to one task identifier. This might occur when merging branches of a conditional statement. Note also that MHP states are multisets because we might have several tasks created by invoking the same method. Since LMHP states are multisets, we write $(q, i) \in E$ to indicate that atom q appears $i > 0$ times in E.

The LMHP analysis of [3], that infers the LMHP states described above, is a data-flow analysis based on the transfer function τ in Fig. 5, except for Case (3) which is novel to our extension and whose auxiliary functions will be given and explained later. Recall that the role of the transfer function in a data-flow analysis is to abstractly execute the different instructions, i.e., transforming one LMHP state to another. Let use explain the relevant cases of τ:

- Case (1) handles method calls, it adds a new active task (an instance of m) that is bound to future variable y, and renames all atoms that already use y to use $\star$ since it is overwritten;
- Case (2) handles **await**, it changes the state of any task bound to future variable y to finished; and
- Case (4) corresponds to other instructions that do not create or wait for tasks to finish. In this case the abstract state is not affected.

In addition, the LMHP analysis merges states of conditional branches using union of multisets, and loops are iterated, with a corresponding widening oper-

ator that transforms unstable MHP atoms (q, i) to (q, ∞), until a fix-point is reached.

Example 1. Consider a method f with a body **while**$(*)$`{ y=o.m();}`. The first time we apply τ over f, we obtain $\{y{:}\mathtt{T}(m, \mathtt{act})\}$ at the exit program point of the while. At the next iteration, we add a new atom bound to y so we lose the association existing in the current state and add the new atom, obtaining $\{\star{:}\mathtt{T}(m, \mathtt{act}), y{:}\mathtt{T}(m, \mathtt{act})\}$. After applying one more iteration, we lose the relation between y and the task m again obtaining $\{(\star{:}\mathtt{T}(m, \mathtt{act}), 2), y{:}\mathtt{T}(m, \mathtt{act})\}$. When comparing the last two LMHP states, we observe that $\star{:}\mathtt{T}(m, \mathtt{act})$ is unstable, thus we apply widening and obtain $\{(\star{:}\mathtt{T}(m, \mathtt{act}), \infty), y{:}\mathtt{T}(m, \mathtt{act})\}$.

In what follows we present how to extend the transfer function τ and the LMHP states to handle returned futures in Case (3). We first explain it using a simple example, and then describe it formally.

Example 2. Assume we have a method f with an instruction "**return** x", and that at the exit program point of f we have a LMHP state $E_0 = \{x{:}\mathtt{T}(h, \mathtt{act}), w{:}\mathtt{T}(g, \mathtt{act})\}$, which means that at the exit program point of f we have two active instances of methods h and g, bound to future variables x and w respectively. This means that f returns a future variable that is bound to an active instance of h. Now assume that in some other method, at some program point, we have a state $E_1 = \{y{:}\mathtt{T}(f, \mathtt{fin}), r{:}\mathtt{T}(k, \mathtt{act}), u{:}\mathtt{T}(l, \mathtt{act})\}$, which means, among other things, that before reaching the corresponding program point, we have invoked f and waited for it to finish (via future variable y). Let us now execute the instruction $u = y.$**get** in the context of E_1 and generate a new LMHP state E_2. Since y is bound to a task that is an instance of f, E_2 should include an atom representing that u is bound to an active task which is an instance of h (which is returned by f via a future variable). Having this information in E_2 allows us to mark h as finished when executing **await** u? later. We do this as follows:

- any MHP atom from E_1 that does not involve u or y is copied to E_2.
- any MHP atom from E_1 that involves u is copied to E_2 but with u renamed to $\star$ because u is overwritten.
- we transfer the atom $x{:}\mathtt{T}(h, \mathtt{act})$ from E_0 to E_2, by adding $u{:}\mathtt{T}(h, \mathtt{act})$ to E_2 since now the corresponding task is bound to u as well.
- the atom $y{:}\mathtt{T}(f, \mathtt{fin})$ must be copied to E_2 as well, but we first rewrite it to $y{:}\mathtt{T}(f, \overline{\mathtt{fin}})$ (in E_2) to indicate that we have incorporated the information from the exit program point of f already. This is important because after executing the **get**, we will have two instances of h in E_0 and E_2 that refer to the same task, and we want to avoid considering them as two different ones in the global phase that we will describe in the next section.

This results in $E_2 = \{y{:}\mathtt{T}(f, \overline{\mathtt{fin}}), r{:}\mathtt{T}(k, \mathtt{act}), \star{:}\mathtt{T}(l, \mathtt{act}), u{:}\mathtt{T}(h, \mathtt{act})\}$.

To summarize the above example, the local phase of our analysis extends that of [3] in two ways: it introduces a new kind of LMHP atom; and it has to treat the **get** instruction in a special way. In the rest of this section we formalize this

extension by providing the auxiliary functions and the data-flow inference. As notation, we let E_ℓ be the LMHP state that corresponds to program point ℓ; we let E^m_{exit} be the LMHP state that corresponds to the exit program point of method m; and we define

$$Ret(m) = \{\mathtt{T}(n, X) \mid \textbf{return}\ y \in body(m),\ y{:}\mathtt{T}(n, X) \in E^m_{exit}\},$$

which is the set of tasks in E^m_{exit} that are bound to a future variable that is returned by method m. This set is needed in order to incorporate these tasks when abstractly executing a **get** instruction as we have seen in the example above. We also let $\mathtt{eliminate}(Y, E)$ be the LMHP set obtained from E by removing all atoms that involve a future variable $y \in Y$. We first modify the transfer function of [3] to treat the instruction $z = y.\textbf{get}$, similarly to what we have done in the example above. This is done by adding Case (3) to the transfer function of Fig. 5:

- The set E' is obtained from E by renaming future variable z to $\star$, since variable z is overwritten, and then eliminating all atoms associated to future variable y (they will be incorporated in E''' below).
- The set E'' consists of new MHP atoms that correspond to futures that are returned by methods to which y is bound. Note that all are now bound to future variable z.
- In E''' we add all atoms bound to y from E but rewritten to mark them as *already been incorporated*.

Due to the new case added to the transfer function, we need to modify the work-flow of the corresponding data-flow analysis in order to backpropagate the information learned from the returned future variables. This is because the LMHP analysis of one method depends on the LMHP states of other methods (via $Ret(m)$ in Case (3) of τ). This means that a method cannot be analyzed independently from the others as in [3], but rather we have to iterate over their analysis results, in the reverse topological order induced by the corresponding call graph, until their corresponding results stabilize.

Example 3. The left column of the table below shows the LMHP states resulting from applying once the τ function to selected program points, the right column shows the result after one iteration of τ over the results in the left column:

E_{11}: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{act})\}$	E_{11}: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{act})\}$
E_{12}: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{fin})\}$	E_{12}: $\{x{:}\mathtt{T}(\mathsf{request}, \mathtt{fin})\}$
E_{13}: $\tau(z = x.\textbf{get}, E_{12})$	E_{13}: $\{x{:}\mathtt{T}(\mathsf{request}, \overline{\mathtt{fin}}), z{:}\mathtt{T}(\mathsf{result}, \mathtt{act})\}$
E_{14}: $\tau(\textbf{await}\ z?, E_{13})$	E_{14}: $\{x{:}\mathtt{T}(\mathsf{request}, \overline{\mathtt{fin}}), z{:}\mathtt{T}(\mathsf{result}, \mathtt{fin})\}$
E_{15}: $E_{14} \cup \{\star{:}\mathtt{T}(\mathsf{sendMessage}, \mathtt{act})\}$	E_{15}: $\{x{:}\mathtt{T}(\mathsf{request}, \overline{\mathtt{fin}}), z{:}\mathtt{T}(\mathsf{result}, \mathtt{fin}), \star{:}\mathtt{T}(\mathsf{sendMessage}, \mathtt{act})\}$
E_{20}: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act})\}$	E_{20}: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act})\}$
E_{21}: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act})\}$	E_{21}: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act})\}$
E_{22}: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act}))\}$	E_{22}: $\{y{:}\mathtt{T}(\mathsf{result}, \mathtt{act}), p{:}\mathtt{T}(\mathsf{inform}, \mathtt{act}))\}$

Let us explain some of the above LMHP states. In the left column, E_{11} corresponds to the state when reaching program point L11, i.e., before executing the statement **await** x?. It includes x:T(request, act) for the active task invoked at L10. The state E_{12} includes the finished task corresponding to the await instruction of the previous program point. E_{13} cannot be solved, as we need the information from state E_{22} (it is required when calculating E''), which has not been computed yet. Something similar happens with the state E_{14}, which cannot be calculated as the state E_{13} has not been totally computed. Atoms y:T(result , act) and p:T(inform, act) appear in state E_{22} for the active tasks invoked at L19 and L20. The state E_{15} includes $\star$:T(sendMessage, act) for the task invoked at L14, which is not bound to any future variable.

In the right column, after one iteration, we observe that most states are not modified except for E_{13}, E_{14} and E_{15}. As for E_{13}, in the previous step we could not obtain the set E'' when analyzing E_{13} because the function τ had not been applied to request (E_{22} had not been computed). Thus, it considered E_{13}: $E' = \{\}$ as there was no task bound to z; $E'' = \{z$:T(result , act)$\}$ and; $E''' = \{y$:T(request, $\overline{\texttt{fin}}$)$\}$. Having E_{13} calculated, E_{14} is computed modifying the state of result to finished and E_{15} is updated with the new information.

4.2 Global MHP

In this section we describe how to use the LMHP information, inferred by the local phase of Sec. 4.1, in order to construct an MHP graph from which an over-approximation of the set of MHP pairs can be extracted. The construction of the MHP graph is different from the one of [3] in that we need to introduce new kind of nodes to reflect the information carried by the new kind of MHP atom y:T(m, $\overline{\texttt{fin}}$). However, the procedure for computing the MHP pairs from the MHP graph is the same. The *MHP graph* of a given program P is a (weighted) directed graph, denoted by $\mathcal{G}_P$, whose nodes are:

- *method nodes*: each method $m \in \texttt{methods}(P)$ contributes 3 nodes $\texttt{act}(m)$, $\texttt{fin}(m)$ and $\overline{\texttt{fin}}(m)$. We use $X(m)$ to refer to a method node without specifying if it corresponds to $\texttt{act}(m)$, $\texttt{fin}(m)$, or $\overline{\texttt{fin}}(m)$.
- *program point nodes*: each program point $\ell \in \texttt{ppoints}(P)$ contributes a node ℓ.
- *return nodes*: each program point $\ell \in \texttt{ppoints}(P)$ that is an exit program point, of some method m, contributes a node $\bar{\ell}$.
- *future variable nodes*: each future variable $y \in \texttt{futures}(P)$ and program point $\ell \in \texttt{ppoints}(P)$ contribute a node ℓ_y (which can be ignored if y does not appear in the corresponding LMHP state of ℓ).

Note that nodes $\overline{\texttt{fin}}(m)$ and $\bar{\ell}$ are particular to our extension, they do not appear in [3] and will be used, as we will see later, to avoid duplicating tasks that are returned to some calling context.

The edges of $\mathcal{G}_P$ are constructed in two steps. First we construct those that do not depend on the LMHP states, and afterwards those that are induced by

LMHP states. The first kind of edges are constructed as follows, for each method $m \in \texttt{methods}(P)$:

- there are edges from $\texttt{act}(m)$ to all *program point nodes* $\ell \in \texttt{ppoints}(m)$. This kind of edges indicate that an active task can be executing at any program point, including its exit program point;
- there is an edge from $\texttt{fin}(m)$ to the exit *program point node* ℓ of m. This kind of edges indicate that a finished task can be only at the exit program point;
- there is an edge from $\overline{\texttt{fin}}(m)$ to the corresponding *return node* $\bar{\ell}$, i.e., ℓ here is the exit program point of m. This kind of edges are similar to the previous ones, but they will be used to avoid duplicating tasks that were returned to some calling context.

All the above edges have weight 0. Next we construct the edges induced by the LMHP states. For each program point $\ell \in \texttt{ppoints}(P)$, we consider E_ℓ and construct the following edges:

- if $(\star{:}\mathtt{T}(m, X), i) \in E_\ell$, we add an edge from node ℓ to node $X(m)$ with weight i. If ℓ is an exit program point we also add an edge from node $\bar{\ell}$ to node $X(m)$ with weight i;
- if $(y{:}\mathtt{T}(m, X), i) \in E_\ell$, we add an edge from node ℓ to node ℓ_y with weight 0 and an edge from node ℓ_y to node $X(m)$ with weight i. In addition, if ℓ is an exit program point and y is not a returned future we add an edge from node $\bar{\ell}$ to node ℓ_y with weight 0.

Note that when ℓ is an exit program point, the difference between node ℓ and $\bar{\ell}$ is that the later ignores tasks that were returned via future variables.

Example 4. Fig. 6 shows the MHP graph for some program points of interest for our running example. Note that the out-going edges of program point nodes in $\mathcal{G}$ coincide with the LMHP states at these program points depicted in Ex. 3. At program point $L15$, the LMHP state E_{15} contains the atoms $x{:}\mathtt{T}(\mathsf{request}, \overline{\texttt{fin}})$, $z{:}\mathtt{T}(\mathsf{result}, \texttt{fin})$ and $\star{:}\mathtt{T}(\mathsf{sendMessage}, \texttt{act})$. Each of these atoms corresponds to one of the edges from program point node 15. The first one is represented by the edge that goes from program point node 15 to future variable node 15_x and from 15_x to method node $\overline{\texttt{fin}}(\mathsf{request})$. The second one corresponds to the edge that goes from 15 to 15_z and from there to method node $\texttt{fin}(\mathsf{result})$. The edge which goes from 15 to method node $\texttt{act}(\mathsf{sendMessage})$ originates from the MHP atom $\star{:}\mathtt{T}(\mathsf{sendMessage}, \texttt{act})$. This last edge does not go to a future variable node as the task is not bound to any future variable ($\star$). Note that we have two nodes 22 and $\overline{22}$ to represent the exit program point L22, connected to $\texttt{fin}(\mathsf{request})$ and $\overline{\texttt{fin}}(\mathsf{request})$. The edges that go out from 22 correspond to the atoms in E_{22}. As L22 is the exit program point of method request, we have to build an edge. This edge goes from $\overline{22}$ to 22_p and from there to $\texttt{act}(\mathsf{inform})$ and corresponds to the atom in E_{22} whose future variable is not returned by request.

Given $\mathcal{G}_P$, using the same procedure as in [3], we say that two program points ℓ_1, ℓ_2 may run in parallel if one of the following conditions hold:

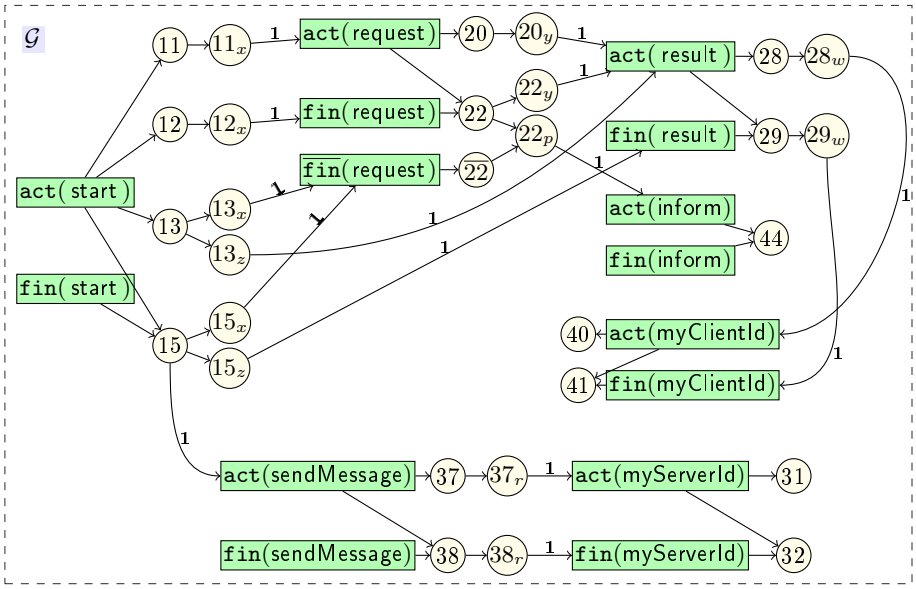


Fig. 6. MHP graph obtained from the analysis of program in Fig. 2.

1. there is a non-empty path from ℓ_1 to ℓ_2 or vice-versa; or
2. there is a program point ℓ_3 and non-empty paths from ℓ_3 to ℓ_1 and from ℓ_3 to ℓ_2 such that the first edge is different, or they share the first edge but it has weight $i > 1$.

The first case is called *direct* MHP pairs and the second one *indirect* MHP pairs.

Example 5. Let us explain some of the MHP pairs shown in Fig. 4 and induced by $\mathcal{G}$. (22,28) and (22,44) are direct MHP pairs as we can find the paths $22 \rightsquigarrow 28$ and $22 \rightsquigarrow 44$ in $\mathcal{G}$. In addition, as the first edge is different, we can conclude that (28,44) is an indirect pair. In contrast to the graph that one would obtain for the original analysis, (15,28) is not an MHP pair (marked in bold in Fig. 4). Instead, we have the path $15 \rightsquigarrow 29$ which indicates that the task result is finished. Similarly, the analysis does not infer the pair (28,37), allowing us to discard the deadlock cycle described in Sec. 3. We find the path $15 \rightsquigarrow 37$ in $\mathcal{G}$, but the path $15 \rightsquigarrow 28$, needed to infer this spurious pair, is not in $\mathcal{G}$.

Let $\tilde{\mathcal{E}}_P$ be the set of MHP pairs obtained by applying the procedure above.

Theorem 1 (soundness). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

5 Implementation and experimental evaluation

The analysis presented in Sec. 4 has been implemented in SACO [2], a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects, which is able to infer deadlock, termination

Examples	Lines	N	PPs^2	OMHPs	MHPs	Lmhp	Gmhp	Mhp	OT	T
ServerClient	69	19	361	176	140	<5	<5	22	64	60
Chat	331	73	5329	1351	1028	<5	<5	606	1686	1014
MailServer	140	28	784	315	284	<5	<5	47	172	166
DistHT	168	27	729	392	367	<5	<5	51	186	183
PeerToPeer	215	42	1764	325	297	8	<5	112	452	238
ETICS	1717	297	88209	30554	30523	175	20	40887	53023	53450
TradingSys^1	1508	216	46656	40562	33038	53	15	13260	32065	31589
TradingSys^2	1508	216	46656	41345	41345	62	26	11765	30308	31057

Fig. 7. Examples and statistics

and resource boundedness [14]. Our analysis has been built on top of the original MHP analysis in SACO and can be tried online at: `http://costa.ls.fi.upm.es/saco/web/` by selecting `MHP` from the menu as type of analysis, then enabling the option `Global Futures Synchronization` in the `Settings` section, and clicking on `Apply`. The benchmarks are also available in the folder `ATVA17`. Given a program with a main procedure, the analysis returns a list of MHP pairs and some statistics about the runtime of the local and global phases.

Fig. 7 summarizes our experiments. The first benchmark ServerClient corresponds to the complete implementation of our running example. The next four are some traditional programs for distributed and concurrent programming: Chat models a chat application, MailServer models a distributed mail server with several users, DistHT implements and uses a distributed hash table and PeerToPeer which represents a peer-to-peer network. The last two examples, ETICS and TradingSys are industrial case studies, respectively, developed by Engineering® and Fredhopper® that model a system for remotely hosting and managing IT resources and a system to manage sales and other facilities on a large product database. These case studies are very conservative on the use of futures (namely only 3 tasks return a future), however, we have included them to assess the efficiency of our analysis on large programs. For the TradingSys, we have two versions, TradingSys^1 which creates a constant number of tasks (namely 3), and TradingSys^2 which creates an unknown number of tasks within a loop. Experiments have been performed on an Intel Core i7-6500U at 2.5GHz x 4 and 7.5GB of Memory, running Ubuntu 16.04. For each program P, $\mathcal{G}_P$ is built and the relation $\tilde{\mathcal{E}}_P$ is computed for those points that affect the concurrency of the program (i.e, entry points of methods, awaits, gets and exit points of methods).

Let us first discuss the accuracy of our approach. Columns **Examples** and **Lines** show the name and number of lines of the benchmark. **N** is the number of program point nodes in $\mathcal{G}_P$. **PPs^2** is the square of the number of program points, i.e., the total number of pairs that could potentially run in parallel. **OMHPs** and **MHPs** show the number of MHP pairs inferred by the original analysis [3] and by ours. **PPs^2-MHPs** is thus the number of MHP pairs that are detected not to happen in parallel by the original analysis. Naturally the original analysis already eliminates many pairs that arise from local future variables (not

returned). **OMHPs-MHPs** gives us the number of further spurious MHP pairs that our analysis eliminates. We can observe that for all examples (except for `TradingSys`^{2}) we reduce the number of inferred MHP pairs (ranging from a small reduction of 0.2% pairs for `ETICS` to a big reduction of 23.9% for `Chat`). In `TradingSys`^{2} we do not eliminate any pair because the tasks created within the loop use the same future variable to return their results, and the analysis needs to over-approximate and assume that all of them may run in parallel.

As regards the efficiency of the analysis, the next three columns contain the time (in milliseconds) taken by the local MHP (**Lmhp**), the graph construction (**Gmhp**) and the time needed to infer the MHP pairs (**Mhp**). The data presented are the average time obtained across several executions. We can observe that both LMHP and the graph construction are very efficient and they only take 0.175s in the largest case. The inference of the MHP pairs is more complex and takes more time. This time depends on the number of program point nodes that the graphs contain. For medium programs, the inference technique is also efficient (taking 0.6s in the largest case), but the time increases notably in bigger examples, reaching 40.8s in our experiments. However, in most applications we are only interested in a subset of pairs. Besides, the pairs can be computed on demand, spending less time to infer them. The last two columns contain the total time (in milliseconds) taken by the analysis of [3] (**OT**) and our approach (**T**). It can be observed that our analysis is more efficient than the original one for all examples except for the `TradingSys`^{2} and `ETICS`, being the overhead negligible in these cases (less than 2.5 %). The reason for the efficiency gain is that when returned futures are tracked, our graph contains less paths that are inspected to infer the MHP pairs. Thus, the process of computing all the feasible paths is faster in these cases, and the global time of the analysis is smaller than [3].

6 Conclusions and related work

An MHP analysis learns from the future variables used in synchronization instructions when tasks are terminated, so that the analysis can accurately eliminate unfeasible MHP pairs that would be otherwise inferred. Some existing MHP analyses [3, 15, 1, 16] for asynchronous programs lose all the information when future variables are awaited in a different scope to the one that spawns the tasks bound to the futures. We have presented a static MHP analysis which captures inter-procedural MHP relations in which future variables are propagated *backwards* from one task to another(s). This implies that a task can be awaited in an outer scope from the one in which it was created. Previous work [5] has considered the propagation of future variables *forward*, i.e., when future variables are passed as arguments of the tasks. This implies that a task can be awaited in an inner scope from the one in which it was created. Also, other MHP analyses allow synchronizing the termination of the tasks in an inner scope, passing them as arguments of methods, namely: [11] considers a fork-join semantics and uses a Happens-Before analysis to infer the MHP information; in [6, 8], programs are abstracted to a thread model which is then analyzed to infer the MHP pairs; [9]

builds a time based model to infer race conditions in high performance systems; this work is extended in [7], using a model checker to solve the MHP decision problem. The last six analyses are imprecise though when future variables or the tasks identifiers are returned by methods and awaited in an outer scope.

The solutions for the backwards and forward inference (namely as formalized in [5]) are technically different, but fully compatible. Essentially, they only have in common that both the local and global analysis phases need to be changed. For the forward inference, the analysis includes a separated *must-have-finished* (MHF) pre-analysis that allows inferring, for each program point ℓ, which tasks (both the tasks spawned locally and the passed as arguments) have finished their execution when reaching ℓ. In contrast, for the backwards inference, the local phase itself has to be extended to propagate backwards the new relations created when a future variable is returned, which requires changing the analysis flow. In both analyses, the creation of the graph needs to be modified to reflect the new information inferred by the respective local phases, but in each case is different. For the forward inference, the way in which the MHP pairs are inferred besides has to be modified. All in all, both extensions are fully compatible, and together provide a full treatment of future variables in the MHP analysis.

References

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Proc. of PPOPP'07*, pages 183–193. ACM, 2007.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
3. E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
4. E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*, LNCS 8172, pages 349–364. Springer, October 2013.
5. E. Albert, S. Genaim, and P. Gordillo. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Proc. of SAS 2015*, volume 9291 of *LNCS*, pages 72–89. Springer, 2015.
6. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *LCPC'05*, LNCS 4339, pages 152–169. Springer, 2005.
7. C. W. Chang and R. Dömer. May-happen-in-parallel analysis of ESL models using UPPAAL model checking. In *DATE 2015*, pages 1567–1570. IEEE, March 2015.
8. C. Chen, W. Huo, L. Li, X. Feng, and K. Xing. Can we make it faster? Efficient may-happen-in-parallel analysis revisited. In *PDCAT 2012*, pages 59–64, Dec 2012.
9. W. Chen, X. Han, and R. Dömer. May-happen-in-parallel analysis based on segment graphs for safe ESL models. In *DATE 2014*, pages 1–6. IEEE, March 2014.
10. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *ESOP'07*, LNCS 4421, pages 316–330. Springer, 2007.

11. P. Di, Y. Sui, D. Ye, and J. Xue. Region-based may-happen-in-parallel analysis for C programs. In *ICPP 2015*, pages 889–898. IEEE, Sept 2015.
12. C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL'95, 22nd ACM SIGPLAN-SIGACT*, 1995.
13. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS 7892, pages 273–288. Springer, 2013.
14. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10 (Revised Papers)*, LNCS 6957, pp. 142-164. Springer, 2012.
15. J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient may happen in parallel analysis for async-finish parallelism. In *In SAS 2012*, volume 7460, pages 5–23. Springer, 2012.
16. A. Sankar, S. Chakraborty, and V. K. Nandivada. Improved mhp analysis. In *CC 2016*, pages 207–217. ACM, Sept 2016.