May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization[⋆]

Elvira Albert, Samir Genaim, and Pablo Gordillo

Complutense University of Madrid (UCM), Spain

Abstract. A may-happen-in-parallel (MHP) analysis computes pairs of program points that may execute *in parallel* across different distributed components. This information has been proven to be essential to infer both safety properties (e.g., deadlock freedom) and liveness properties (e.g., termination and resource boundedness) of asynchronous programs. Existing MHP analyses take advantage of the synchronization points to learn that one task has finished and thus will not happen in parallel with other tasks that are still active. Our starting point is an existing MHP analysis developed for *intra-procedural* synchronization, i.e., it only allows synchronizing with tasks that have been spawned inside the current task. This paper leverages such MHP analysis to handle *inter-procedural* synchronization, i.e., a task spawned by one task can be awaited within a different task. This is challenging because task synchronization goes beyond the boundaries of methods, and thus the inference of MHP relations requires novel extensions to capture inter-procedural dependencies. The analysis has been implemented and it can be tried online.

1 Introduction

In order to improve program performance and responsiveness, many modern programming languages and libraries promote an asynchronous programming model, in which *asynchronous* tasks can execute concurrently with their caller tasks, and their callers can explicitly wait for their completion. Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables for synchronization [8, 7]. In this model, a method call m on some parameters $\overline{x}$, written as f=m($\overline{x}$), spawns an asynchronous task. Here, f is a *future variable* which allows synchronizing with the termination of the task executing m. The instruction **await** f? allows checking whether m has finished, and blocks the execution of the current task if m is still running. As concurrently-executing tasks interleave their accesses to shared memory, asynchronous programs are prone to concurrency-related errors [6]. Automatically proving safety and liveness properties still remains a challenging endeavor today.

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

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 parallel across different distributed components. 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 **await** instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the **await**. This piece of information is fundamental to prove more complex properties: in [9], MHP pairs are used to discard unfeasible deadlock cycles; in [4], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. As a simple example, consider a procedure `g` that contains as unique instruction `y=−1`, where `y` is a global variable. The following loop `y=1; while(i>0){i=i−y;}` might not terminate if `g` runs in parallel with it, since `g` can modify `y` to a negative value and the loop counter will keep on increasing. However, if we can guarantee that `g` will not run in parallel with this code, we can ensure termination and resource-boundedness for the loop.

This paper leverages an existing MHP analysis [3] developed for intra-procedural synchronization to the more general setting of inter-procedural synchronization. This is a fundamental extension because it allows synchronizing with the termination of a task outside the scope in which the task is spawned, as it is available in most concurrent languages. In the above example, if task `g` is awaited outside the boundary of the method that has spawned it, the analysis of [3] assumes that it may run in parallel with the loop and hence it fails to prove termination and resource boundedness. The enhancement to inter-procedural synchronization requires the following relevant extensions to the analysis:

1. *Must-have-finished analysis* (MHF): the development of a novel MHF analysis which infers *inter-procedural dependencies* among the tasks. Such dependencies allow us to determine that, when a task finishes, those that are awaited for on it must have finished as well. The analysis is based on using Boolean logic to represent abstract states and simulate corresponding operations. The key contribution is the use of logical implication to delay the incorporation of procedure summaries until synchronization points are reached. This addresses a challenge in the analysis of asynchronous programs.
2. *Local MHP phase*: the integration of the above MHF information in the local phase of the original MHP analysis in which methods are analyzed locally, i.e., without taking indirect calls into account. This will require the use of richer analysis information in order to consider the inter-procedural dependencies inferred in point 1 above.
3. *Global MHP phase*: the refinement of the global phase of the MHP analysis –where the information of the local MHP analysis in point 2 is composed– in order to eliminate spurious MHP pairs which appear when inter-procedural dependencies are not tracked. This will require to refine the way in which MHP pairs are computed.

We have implemented our approach in SACO [2], a static analyzer for concurrent objects which is able to infer the aforementioned liveness and safety properties. The system can be used online at `http://costa.ls.fi.upm.es/saco/web`, where the examples used in the paper are also available.

2 Language

Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables [8, 7]. It also includes conditional and loop constructs, however, conditions in these constructs are simply non-deterministic choices. Developing the analysis at such abstract level is convenient [11], since the actual computations are simply ignored in the analysis and what is actually tracked is the control flow that originates from asynchronously calling methods and synchronizing with their termination. Our implementation, however, is done for the full concurrent object-oriented language ABS [10] (see Sec. 6).

A program P is a set of methods that adhere to the following grammar:

$$
\begin{array}{lll}
M & ::= & m(\bar{x})\ \{s\} \\
s & ::= & \epsilon \mid b; s \\
b & ::= & \textsf{if}\ (*)\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{while}\ (*)\ \textsf{do}\ s \mid y = m(\bar{x}) \mid \textsf{await}\ x? \mid \textsf{skip}
\end{array}
$$

Here all variables are future variables, which are used to synchronize with the termination of the called methods. Those future variables that are used in a method but are not in its parameters are the *local future variables* of the method (thus we do not need any special instruction for declaring them). In loops and conditions, the symbol $*$ stands for non-deterministic choice (*true* or *false*). The instruction $y = m(\bar{x})$ creates a new task which executes method m, and binds the future variable y with this new task so we can synchronize with its termination later. Inter-procedural synchronization is realized in the language by passing future variables as parameters, since the method that receives the future variable can await for the termination of the associated task (created outside its scope). For simplifying the presentation, we assume that *method parameters are not modified inside each method.* For a method m, we let P_m be the set of its parameters, L_m the set of its local variables, and $V_m = P_m \cup L_m$.

The instruction **await** $x?$ blocks the execution of the current task until the task associated with x terminates. Instruction **skip** has no effect, it is simply used when abstracting from a richer language, e.g., ABS in our case, to abstract instructions such as assignments. Programs should include a method main from which the execution (and the analysis) starts. We assume that instructions are labeled with unique identifiers that we call program points. For **if** and **while** the identifier refers to the corresponding condition. We also assume that each method has an exit program point ℓ_m. We let $\texttt{ppoints}(m)$ and $\texttt{ppoints}(P)$ be the sets of program points of method m and program P, resp., I_ℓ be the instruction at program point ℓ, and $\texttt{pre}(\ell)$ be the set of program points preceding ℓ.

Next we define a formal (interleaving) operational semantics for our language. A task is of the form $tsk(tid, l, s)$ where tid is a unique identifier, l is a mapping

$$(\textsc{skip})\quad \frac{}{tsk(tid, l, \textsf{skip}; s) \rightsquigarrow tsk(tid, l, s)}$$

$$(\textsc{if})\quad \frac{b \equiv \textsf{if}\ (*)\ \textsf{then}\ s_1\ \textsf{else}\ s_2,\ \text{set } s' \text{ non-deterministically to } s_1; s \text{ or } s_2; s}{tsk(tid, l, b; s) \rightsquigarrow tsk(tid, l, s')}$$

$$(\textsc{loop})\quad \frac{b \equiv \textsf{while}\ (*)\ \textsf{do}\ s_1,\ \text{set } s' \text{ non-deterministically to } s_1; b; s \text{ or } s}{tsk(tid, l, b; s) \rightsquigarrow tsk(tid, l, s')}$$

$$(\textsc{call})\quad \frac{\bar{z} \text{ are the formal parameters of } m,\ tid' \text{ is a fresh id},\ l' = \{z_i \mapsto l(x_i)\}}{tsk(tid, l, y = m(\bar{x}); s) \rightsquigarrow tsk(tid, l[y \mapsto tid'], s), tsk(tid', l', body(m))}$$

$$(\textsc{await})\quad \frac{l(x) = tid'}{tsk(tid, l, \textsf{await}\ x?; s), tsk(tid', l', \epsilon) \rightsquigarrow tsk(tid, l, s), tsk(tid', l', \epsilon)}$$

Fig. 1. Derivation Rules

from local variables and parameters to task identifiers, and s is a sequence of instructions. Local futures are initialized to the special value $\perp$ which is the default value for future variable (i.e., $\perp$ like **null** for reference variables in Java). A state S is a set of tasks that are executing in parallel. From a state S we can reach a state S' in one execution step, denoted $S \rightsquigarrow S'$, if S can be rewritten using one of the derivation rules of Fig. 1 as follows: if the conclusion of the rule is $A \rightsquigarrow B$ such that $A \subseteq S$ and the premise holds, then $S' = (S \setminus A) \cup B$. The meaning of the derivation rules is quite straightforward: (SKIP) advances the execution of the corresponding task to the next instruction; (IF) nondeterministically chooses between one of the branches; (LOOP) nondeterministically chooses between executing the loop body or advancing to the instruction after the loop; (CALL) creates a new task with a fresh identifier tid', initializes the formal parameters $\bar{z}$ of m to those of the actual parameters $\bar{x}$, sets future variable y in the calling task to tid', so one can synchronize with its termination later (other local futures of m are assumed to have the special value $\perp$); and (AWAIT) advances to the next instruction if the task associated to x has terminated already. Note that when a task terminates, it does not disappear from the state but rather its sequence of instructions remains empty.

An execution is a sequence of states $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_n$, sometimes denoted as $S_0 \rightsquigarrow^* S_n$, where $S_0 = \{tsk(0, l, body(\mathsf{main}))\}$ is an initial state which includes a single task that corresponds to method main, and l is an empty mapping. At each step there might be several ways to move to the next state depending on the task selected, and thus executions are nondeterministic.

In what follows, given a task $tsk(tid, l, s)$, we let $pp(s)$ be the program point of the first instruction in s. When s is an empty sequence, $pp(s)$ refers to the exit program point of the corresponding method. Given a state S, we define its set of MHP pairs, i.e., the set of program points that execute in parallel in S, as $\mathcal{E}(S) = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, l_1, s_1), tsk(tid_2, l_2, s_2) \in S, tid_1 \neq tid_2\}$. The set of MHP pairs for a program P is then defined as the set of MHP pairs of all reachable states, namely $\mathcal{E}_P = \cup\{\mathcal{E}(S_n) \mid S_0 \leadsto^* S_n\}$.

Example 1. Fig. 2 shows some examples in our language, where m_1, m_2 and m_3 are main methods. The following are some steps in a possible derivation for m_2:

$$\begin{array}{l}
\mathbf{S_0} \equiv tsk(0, \emptyset, body(\mathsf{m}_2)) \leadsto^* \mathbf{S_1} \equiv tsk(0, [x \mapsto 1], \{16, \ldots\}), tsk(1, \emptyset, body(\mathsf{f})) \leadsto^* \\
\mathbf{S_2} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{18, \ldots\}), tsk(1, \emptyset, body(\mathsf{f})), tsk(2, [w \mapsto 1], body(\mathsf{g})) \leadsto^* \\
\mathbf{S_3} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{19, \ldots\}), tsk(1, \emptyset, \epsilon), tsk(2, [w \mapsto 1], body(\mathsf{g})) \leadsto^* \\
\mathbf{S_4} \equiv tsk(0, [x \mapsto 1, z \mapsto 2], \{20, \ldots\}), tsk(1, \emptyset, \epsilon), tsk(2, [w \mapsto 1], \epsilon) \leadsto \ldots
\end{array}$$

In S_1 we execute until the asynchronous call to f which creates a new task identified as 1 and binds x to this new task. In S_2 we have executed the **skip** and the asynchronous invocation to g that adds in the new task the binding of the formal parameter w to the task identified as 1. In S_3 we proceed with the execution of the instructions in m_2 until reaching the **await** that blocks this task until g terminates. Also, in S_3 we have executed entirely f (denoted by ϵ). S_4 proceeds with the execution of g whose **await** can be executed since task 1 is at its exit point ϵ. We have the following MHP pairs in this fragment of the derivation, among many others: from S_1 we have (16,35) that captures that the first instruction of f executes in parallel with the instruction 16 of m2, from S_2 we have (18,35) and (18,38). The important point is that we have no pair (20,35) since when the **await** at L19 executes at S_4, it is guaranteed that f has finished. This is due to the inter-procedural dependency at L39 of g where the task f is awaited: variable x is passed as argument to g, which allows g to synchronize with the termination of f at L39 even if f was called in a different method.

3 An Informal Account of our Method

In this section, we provide an overview of our method by explaining the analysis of m_2. Our goal is to infer precise MHP information that describes, among others, the following representative cases: (1) any program point of g cannot run in parallel with L20, because at L19 method m_2 awaits for g to terminate; (2) L35 cannot run in parallel with L20, since when waiting for the termination of g at L19 we know that f *must-have-finished* as well due to the *dependency* relation that arises when m_2 implicitly waits for the termination of f; and (3) L35 cannot run in parallel with L40, because f *must-have-finished* due to the synchronization on the local future variable w at L39 that refers to future variable x of m_2.

Let us first informally explain which MHP information the analysis of [3] is able to infer for m_2, and identify the reasons why it fails to infer some of the desired information. The analysis of [3] is carried out in two phases: (1) each method is *analyzed separately* to infer local MHP information; and (2) the local information is used to construct a global MHP graph from which MHP pairs are extracted by checking reachability conditions among the nodes.

The local analysis infers, for each program point, a *multiset* of MHP atoms where each atom describes a task that might be executing in parallel when

```
1  m1() {
2    x=f();
3    z=q();
4    skip
5    if (*) then
6      w=g(x);
7      skip;
8    else
9      w=k(x,z);
10     skip;
11   await w?;
12 }
```

```
13 m2() {
14   skip;
15   x=f();
16   skip;
17   z=g(x);
18   skip;
19   await z?;
20   skip;
21 }
22
23
24
```

```
25 m3() {
26   z=f();
27   while (*)
28     x=q();
29     w=h(x,z);
30     await w?;
31     skip;
32 }
33
34 f() {
35   skip;
36 }
```

```
37 g(w) {
38   skip;
39   await w?
40   skip;
41 }
42
43 h(a,b) {
44   skip;
45   z=g(a);
46   skip;
47   await z?;
48 }
```

```
49 k(a,b) {
50   skip;
51   await a?;
52   skip;
53   await b?;
54   skip;
55 }
56
57 q() {
58   skip;
59 }
60
```

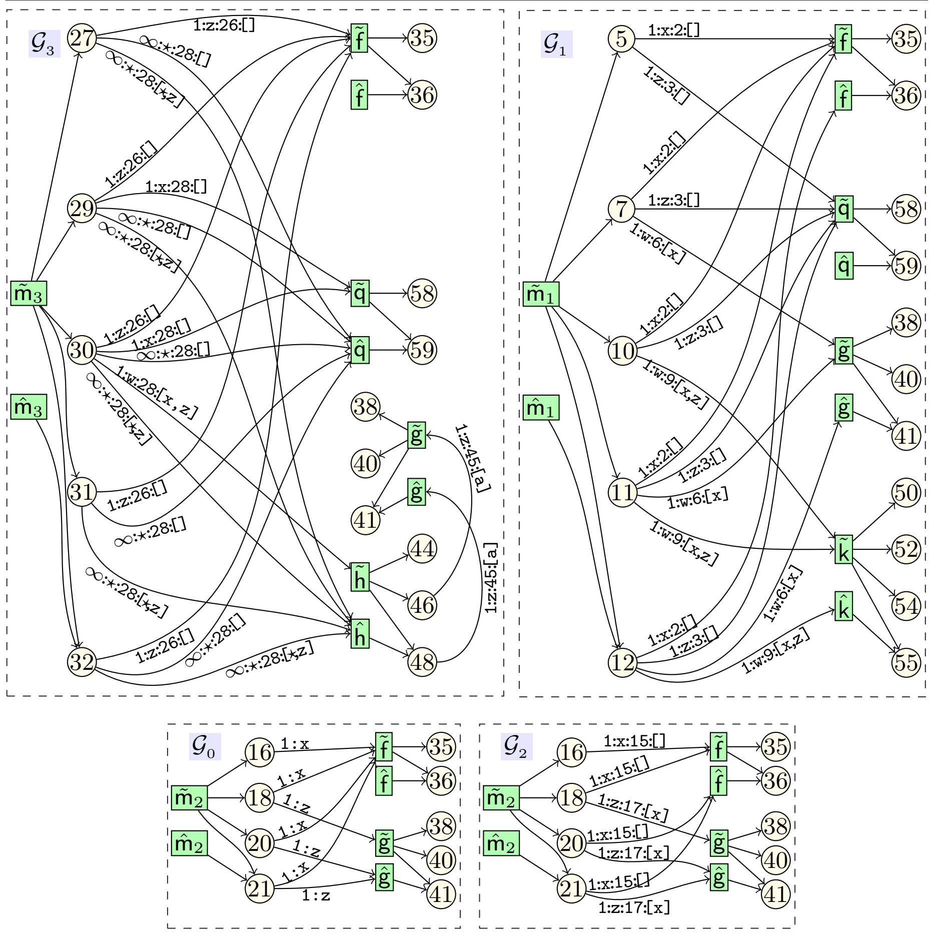

Fig. 2. (TOP) Examples for MHP analysis (m_1, m_2, m_3 are main methods). (BOTTOM) MHP graph $\mathcal{G}_i$ corresponds to analyzing m_i, and $\mathcal{G}_0$ to analyzing m_2 as in [3].

reaching that program point, but only considering tasks that have been invoked directly in the analyzed method. An atom of the form $x{:}\tilde{m}$ indicates that there might be an *active* instance of m executing at any of its program points, and is bound to the future variable x. An atom of the form $x{:}\hat{m}$ differs from the previous one in that m must be at its exit program point, i.e., has finished executing already. For method m_2, the local MHP analysis infers, among others, $\{\mathsf{x}{:}\tilde{\mathsf{f}}\}$ for L16, $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\tilde{\mathsf{g}}\}$ for L18, and $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\hat{\mathsf{g}}\}$ for L20 and L21, because g has been awaited locally. Observe that the sets of L20 and L21 include $\mathsf{x}{:}\tilde{\mathsf{f}}$ and not $\mathsf{x}{:}\hat{\mathsf{f}}$, although method f has finished already when reaching L20 and L21 (since g has finished). This information cannot be inferred by the local analysis of [3] since it is applied to each method separately, ignoring (a) indirect (non-local) calls and (b) inter-procedural synchronizations. In the sequel we let Ψ_ℓ be the result of the local MHP analysis for program point ℓ.

In the second phase, the analysis of [3] builds an MHP graph whose purpose is to capture MHP relations due to indirect calls (point (a) above). The graph $\mathcal{G}_0$ depicted in Fig. 2 for m_2 is constructed as follows: (1) every program point ℓ contributes a node labeled with ℓ – for simplicity we include only program points of interest; (2) every method m contributes two nodes $\tilde{m}$ and $\hat{m}$, where $\tilde{m}$ is connected to all program point nodes of m to indicate that when m is active, it can be executing at any of its program points, and $\hat{m}$ is connected only to the exit program point of m; and (3) if $x{:}\tilde{m}$ (resp. $x{:}\hat{m}$) is an atom of Ψ_ℓ with multiplicity i, i.e., it appears i times in the multiset Ψ_ℓ, we create an edge from ℓ to $\tilde{m}$ (resp. $\hat{m}$) and label it with $i{:}x$. Note such edge actually represents i identical edges, i.e., we could copy the edge i times and omit the label i.

Roughly, the MHP pairs are obtained from $\mathcal{G}_0$ using the following principle: program points (ℓ_1, ℓ_2) might execute in parallel if there is a path from ℓ_1 to ℓ_2 or vice versa (direct MHP pair); or if there is a program point ℓ_3 such that there are paths from ℓ_3 to ℓ_1 and to ℓ_2 (indirect MHP pair), and the first edge of both paths is labeled with two different future variables. When two paths are labeled with the same future variable, it is because there is a disjunction (e.g., from an if-then-else) and only one of the paths might actually occur. Applying this principle to $\mathcal{G}_0$, we can conclude that L20 cannot execute in parallel with any program point of g, which is precise as expected, and that L20 can execute in parallel with L35 which is imprecise. This imprecision is attributed to the fact that the MHP analysis of [3] does not track inter-method synchronizations.

In order to overcome the imprecision, we develop a must-have-finished analysis that captures inter-method synchronizations, and use it to improve the two phases of [3]. This analysis would infer, for example, that "*when reaching* L40*, it is guaranteed that whatever task bound to* w *has finished already*", and that "*when reaching* L20*, it is guaranteed that whatever tasks bound to* x *and* z *have finished already*". By having this information at hand, the first phase of [3] can be improved as follows: when analyzing the effect of **await** z? at L20, we change the status of both g and f to finished, because we know that any task bound z and x has finished already. In addition, we modify the MHP atoms as follows: an MHP atom will be of the form $y{:}\ell{:}\tilde{m}(\bar{x})$ or $y{:}\ell{:}\hat{m}(\bar{x})$, where the new information ℓ and $\bar{x}$ are the calling site and the parameters passed to m. The need for this extra information will become clear later in this section. In summary, the modified first phase will infer $\{$x:15:$\tilde{\mathsf{f}}$()$\}$ for L16, $\{$x:15:$\tilde{\mathsf{f}}$(), z:17:$\tilde{\mathsf{g}}$(x)$\}$ for L18, and $\{$x:15:$\hat{\mathsf{f}}$(), z:17:$\hat{\mathsf{g}}$(x)$\}$ for L20 and L21.

In the second phase of the analysis: (i) the construction of the MHP graph is modified to use the new local MHP information; and (ii) the principle used to extract MHP pairs is modified to make use to the must-have-finished information. The new MHP graph constructed for m_2 is depicted in Fig. 2 as $\mathcal{G}_2$. Observe that the labels on the edges include the new information available in the MHP atoms. Importantly, the spurious MHP information that is inferred by [3] is not included in this graph: (1) in contrast to $\mathcal{G}_0$, $\mathcal{G}_2$ does not include edges from nodes 20 and 21 to $\tilde{\mathsf{f}}$, but to $\hat{\mathsf{f}}$. This implies that L35 cannot run in parallel with L20 or L21; (2) in $\mathcal{G}_2$, we still have paths from 18 to 35 and 40, which means, if the old principle for extracting MHP pairs is used, that L35 and L40 might happen in parallel. The main point is that, using the labels on the edges, we know that the first path uses a call to f that is bound to x, and that this same x is passed to g, using the parameter w, in the first edge of the second path. Now since the must-have-finished analysis tell us that at L40 any task bound w is finished already, we conclude that f must be at its exit program point when the execution reaches L40, and thus the MHP pair (35,40) is spurious because L35 is not an exit program point of f. This last point explains why the MHP atoms are designed to include the actual parameters of method calls.

4 Must-Have-Finished Analysis

In this section we present a novel inter-procedural Must-Have-Finished (MHF) analysis that can be used to compute, for each program point ℓ, a set of *finished future variables*, i.e., whenever ℓ is reached those variables are either not bound to any task (i.e., have the default value $\perp$) or their bound tasks are guaranteed to have terminated. We refer to such sets as MHF sets.

Example 2. The following are MHF sets for the program points of Fig. 2:

L2: {x,w,z}	L9 : {w}	L16: {z}	L26: {x,z,w}	L32: {x,w}	L41: {w}	L50: {}	L58: {}
L3: {z,w}	L10: {}	L17: {z}	L27: {x,w}	L35: {}	L44: {z}	L51: {}	L59: {}
L4: {w}	L11: {}	L18: {}	L28: {x,w}	L36: {}	L45: {z}	L52: {a}	
L5: {w}	L12: {x,w}	L19: {}	L29: {w}	L38: {}	L46: {}	L53: {a}	
L6: {w}	L14: {x,z}	L20: {x,z}	L30: {}	L39: {}	L47: {}	L54: {a,b}	
L7: {}	L15: {x,z}	L21: {x,z}	L31: {x,w}	L40: {w}	L48: {a,z}	L55: {a,b}	

Here, at program points that correspond to method entries, all local variables (but not the parameters) are finished since they point to no task. For g: at L38 and L39 no task is guaranteed to have finished, because the task bound to w might be still executing; at L40 and L41, since we passed through **await** w? already, it is guaranteed that w is finished. For k: at L50 and L51 no task is guaranteed to have finished; at L52 and L53 a is finished since we already passed through **await** a?; and at L54 and L55 both a and b are finished. For m_1: at L12 both w and x are finished. Note that w is finished because of **await** w?, and x is finished due to the implicit dependency between the termination of x and w.

4.1 Definition of MHF

By carefully examining the MHF sets of Ex. 2, we can see that an analysis that simply tracks MHF sets would be imprecise. For example, since the MHF set at L11 is empty, the only information we can deduce for L12 is that w is finished. To deduce that x is finished we must track the implicit dependency between w and x. Next we define a more general MHF property that captures such dependencies, and from which we can easily compute the MHF sets.

Definition 1. *Given a program point* $\ell \in \texttt{ppoints}(P)$*, we let* $\mathcal{F}(\ell) = \{f(S_i, l) \mid S_0 \leadsto^* S_i, tsk(tid, l, s) \in S_i, pp(s) = \ell\}$ *where* $f(S, l) = \{x \mid x \in dom(l),\ l(x) = \bot \vee (l(x) = tid' \wedge tsk(tid', l', \epsilon) \in S)\}$.

Intuitively, $f(S, l)$ is the set of all future variables, from those defined in l, whose corresponding tasks are finished in S. The set $\mathcal{F}(\ell)$ considers all possible ways of reaching ℓ, and for each one it computes a corresponding set $f(S, l)$ of finished future variables. Thus, $\mathcal{F}(\ell)$ describes all possible sets of finished future variables when reaching ℓ. The set of *all* finished future variables at ℓ is then defined as $\texttt{mhf}(\ell) = \cap\{F \mid F \in \mathcal{F}(\ell)\}$, i.e., the intersection of all sets in $\mathcal{F}(\ell)$.

Example 3. The values of $\mathcal{F}(\ell)$ for selected program points from Fig. 2 are:

L5 : {{w,x,z},{w,z},{w,x},{w}}	L31: {{w,x,z},{w,x}}	L46: {{a,z},{a},{},
L11: {{w,x,z},{w,x},{x,z},{z},{x},{}}	L32: {{w,x,z},{w,x}}	{a,b,z},{a,b},{b}}
L12: {{w,x,z},{w,x}}	L35: {{}}	L48: {{a,z},{a,b,z}}
L20: {{x,z}}	L38: {{w},{}}	L52: {{a},{a,b}}
L27: {{w,x,z},{w,x}}	L40: {{w}}	L54: {{a,b}}
L30: {{w,x,z},{w,x},{x,z},{x},{z},{}}		L58: {{}}

In L5 different sets arise by considering all possible orderings in the execution of tasks f, q and m_1, but $\texttt{mhf}(L5) = \{\mathsf{w}\}$. Note that for any $F \in \mathcal{F}(11)$, if $\mathsf{w} \in F$ then $\mathsf{x} \in F$, which means that if w is finished at L11, then x must have finished.

4.2 An Analysis to Infer MHF Sets

Our goal is to infer $\texttt{mhf}(\ell)$, or a subset of it, for each $\ell \in \texttt{ppoints}(P)$. Note that any set X that over-approximates $\mathcal{F}(\ell)$, i.e., $\mathcal{F}(\ell) \subseteq X$, can be used to compute a subset of $\texttt{mhf}(\ell)$, because $\cap\{F \mid F \in X\} \subseteq \cap\{F \mid F \in \mathcal{F}(\ell)\}$. In the rest of this section we develop an analysis to over-approximate $\mathcal{F}(\ell)$. We will use Boolean formulas, whose models naturally represent MHF sets, and Boolean connectives to smoothly model the abstract execution of the different instructions.

An MHF state for the program points of a method m is a propositional formula $\Phi : V_m \mapsto \{true, false\}$ of the form $\vee_i \wedge_j c_{ij}$, where an atomic proposition c_{ij} is either x or $y \to x$ such that $x \in V_m \cup \{true, false\}$ and $y \in L_m$. Intuitively, an atomic proposition x states that x is finished, and $y \to x$ states that if y is finished then x is finished as well. Note that we do not allow the parameters of m to appear in the premise of an implication (we require $y \in L_m$). When Φ is *false* or of the form $\vee_j \wedge_j x_{ij}$ where x_{ij} is a propositional variable, we call it monotone. Recall that $\sigma \subseteq V_m$ is a *model* of Φ, iff an assignment that maps variables from σ to *true* and other variables to *false* is a satisfying assignment for Φ. The set of all models of Φ is denoted $[\![\Phi]\!]$. The set of all MHF states for m, together with the formulas *true* and *false*, is denoted $\mathcal{A}_m$.

Example 4. Assume $V_m = \{x, y, z\}$. The Boolean formula $x \vee y$ states that either x or y or both are finished, and that z can be in any status. This information is precisely captured by the models $[\![x \vee y]\!] = \{\{x\},\{y\},\{x,y\},\{x,z\},\{y,z\},\{x,y,z\}\}$. The Boolean formula $z \wedge (x \rightarrow y)$ states that z is finished, and if x is finished then y is finished. This is reflected in $[\![z \wedge (x \rightarrow y)]\!] = \{\{z\}, \{z, y\}, \{z, x, y\}\}$ since z belongs to all models, and any model that includes x includes y as well. The formula *false* means that the corresponding program point is not reachable. The following MHF states correspond to some selected program points from Fig. 2:

Φ_5 : w	Φ_{12}: w $\wedge$ x	Φ_{27}: w $\wedge$ x	Φ_{31}: w $\wedge$ x	Φ_{35}: *true*	Φ_{40}: w	Φ_{48}: a $\wedge$ z	Φ_{54}: a $\wedge$ b
Φ_{11}: w$\rightarrow$x	Φ_{20}: x $\wedge$ z	Φ_{30}: w$\rightarrow$x	Φ_{32}: w $\wedge$ x	Φ_{38}: *true*	Φ_{46}: z$\rightarrow$a	Φ_{52}: a	Φ_{58}: *true*

Note that the models $[\![\Phi_\ell]\!]$ coincide with $\mathcal{F}(\ell)$ from Ex. 3.

Now, we proceed to explain how the execution of the different instructions can be modeled with Boolean formulas. Let us first define some auxiliary operations. Given a variable x and an MHF state $\Phi \in \mathcal{A}_m$, we let $\exists x.\Phi = \Phi[x \mapsto \mathit{true}] \vee \Phi[x \mapsto \mathit{false}]$, i.e., this operation eliminates variable x from (the domain of) Φ. Note that $\exists x.\Phi \in \mathcal{A}_m$ and that $[\![\Phi]\!] \models [\![\exists x.\Phi]\!]$. For a tuple of variables $\bar{x}$ we let $\exists \bar{x}.\Phi$ be $\exists x_1.\exists x_2.\dots.\exists x_n.\Phi$, i.e., eliminate all variables $\bar{x}$ from Φ. We also let $\bar{\exists} \bar{x}.\Phi$ stand for eliminating all variables but $\bar{x}$ from Φ. Note that if $\Phi \in \mathcal{A}_m$ is monotone, and $x \in L_m$, then $x \rightarrow \Phi$ is a formula in $\mathcal{A}_m$ as well.

Given a program point ℓ, an MHF state Φ_ℓ, and an instruction to execute I_ℓ, our aim is to compute a new MHF state, denoted $\mu(I_\ell)$, that represents the effect of executing I_ℓ within Φ_ℓ. If I_ℓ is **skip**, then clearly $\mu(I_\ell) \equiv \Phi_\ell$. If I_ℓ is an **await** x? instruction, then $\mu(I_\ell)$ is $x \wedge \Phi_\ell$, which restricts the MHF state of Φ_ℓ to those cases (i.e., models) in which x is finished. If I_ℓ is a call $y = m(\bar{x})$, where m is a method with parameters named $\bar{z}$, and, at the exit program point of m we know that the MHF state Φ_{ℓ_m} holds, then $\mu(I_\ell)$ is computed as follows:

- We compute an MHF state Φ_m that describes "what happens to tasks bound to $\bar{x}$ when m terminates". This is done by projecting Φ_{ℓ_m} onto the method parameters, and then renaming the formal parameters $\bar{z}$ to the actual parameters $\bar{x}$, i.e., $\Phi_m = (\bar{\exists} \bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]$, where $[\bar{z}/\bar{x}]$ denotes the renaming.
- Now assume that ξ is a new (future) variable to which m is bound. Then $\xi \rightarrow \Phi_m$ states that "when m terminates, Φ_m must hold". Note that it says nothing about $\bar{x}$ if m has not terminated yet. It is also important to note that Φ_m is monotone and thus $\xi \rightarrow \Phi_m$ is a valid MHF state.
- Next we add $\xi \rightarrow \Phi_m$ to Φ_ℓ, eliminate (old) y since the variable is rewritten, and rename ξ to (new) y. Note that we use ξ as a temporary variable just not to conflict with the old value of y.

The above reasoning is equivalent to $(\exists y.(\Phi_\ell \wedge (\xi \rightarrow (\bar{\exists} \bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]))[\xi/y]$, and is denoted by $\oplus(\Phi_\ell, y, \Phi_{\ell_m}, \bar{x}, \bar{z})$. Note that the use of logical implication $\rightarrow$, to abstractly simulate method calls, allows delaying the incorporation of the method summary Φ_m until corresponding synchronization point is reached.

Example 5. Let $\Phi_{11} = \mathsf{x} \rightarrow \mathsf{w}$ be the MHF state at L11. The effect of executing I_{11}, i.e., **await** w?, within Φ_{11} should eliminate all models that do not include w. This is done using $\mathsf{w} \wedge \Phi_{11}$ which results in $\Phi_{12} = \mathsf{w} \wedge \mathsf{x}$. Now let $\Phi_{29} = \mathsf{w}$ be the MHF state at L29. The effect of executing the instruction at L29, i.e., w=h(x,z), within Φ_{29} is defined as $\oplus(\Phi_{29}, \mathsf{w}, \Phi_{48}, \langle \mathsf{x}, \mathsf{z} \rangle, \langle \mathsf{a}, \mathsf{b} \rangle)$ and computed as follows: (1) we restrict $\Phi_{48} = \mathsf{a} \wedge \mathsf{z}$ to the method parameters $\langle \mathsf{a}, \mathsf{b} \rangle$, which results in a; (2) we rename the formal parameters $\langle \mathsf{a}, \mathsf{b} \rangle$ to the actual ones $\langle \mathsf{x}, \mathsf{z} \rangle$ which results in $\Phi_{\mathsf{h}} = \mathsf{x}$; (3) we compute $\exists \mathsf{w}.(\Phi_{29} \wedge (\xi \rightarrow \Phi_{\mathsf{h}}))$, which results in $\xi \rightarrow \mathsf{x}$; and finally (4) we rename ξ to w which results in $\Phi_{30} = \mathsf{w} \rightarrow \mathsf{x}$.

Next we describe how to generate a set of data-flow equations whose solutions associate to each $\ell \in \mathtt{ppoints}(P)$ an MHF state Φ_ℓ that over-approximates $\mathcal{F}(\ell)$, i.e., $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$. Each $\ell \in \mathtt{ppoints}(P)$ contributes one equation as follows:

- if ℓ is not a method entry, we generate $\Phi_\ell = \vee\{\mu(\ell') \mid \ell' \in \mathtt{pre}(\ell)\}$. This considers each program point ℓ' that immediately precedes ℓ, computes the effect $\mu(\ell')$ of executing $I_{\ell'}$ within $\Phi_{\ell'}$, and the takes their disjunction;
- if ℓ is an entry of method m, we generate $\Phi_\ell = \wedge\{x \mid x \in L_m\}$, i.e., all local variables point to finished tasks (since they are mapped to $\perp$ when entering a method), and we do not know anything about the parameters.

The set of all equations for a program P is denoted by $\mathcal{H}_P$.

Example 6. The following are the equations for the program points of m_3:

$\Phi_{27} = \oplus(\Phi_{26}, \mathsf{z}, \Phi_{36}, \langle\rangle, \langle\rangle) \vee \Phi_{31}$	$\Phi_{28} = \Phi_{27}$	$\Phi_{29} = \oplus(\Phi_{28}, \mathsf{x}, \Phi_{59}, \langle\rangle, \langle\rangle)$	$\Phi_{31} = \mathsf{w} \wedge \Phi_{30}$
$\Phi_{30} = \oplus(\Phi_{29}, \mathsf{w}, \Phi_{48}, \langle \mathsf{x}, \mathsf{z} \rangle, \langle \mathsf{a}, \mathsf{b} \rangle)$		$\Phi_{26} = \mathsf{w} \wedge \mathsf{x} \wedge \mathsf{z}$	$\Phi_{32} = \Phi_{27}$

Note the circular dependency of Φ_{27} and Φ_{31} which originates from the corresponding **while** loop. Recall that m_3 is a main method.

The next step is to solve $\mathcal{H}_P$, i.e., compute an MHF state Φ_ℓ, for each $\ell \in \mathtt{ppoints}(P)$, such that $\mathcal{H}_P$ is satisfiable. This can be done iteratively as follows. We start from an initial solution where $\Phi_\ell = \mathit{false}$ for each $\ell \in \mathtt{ppoints}(P)$. Then repeat the following until a fixed-point is reached: (1) substitute the current solution in the right hand side of the equations, and obtain new values for each Φ_ℓ; and (2) merge the new and old values of each Φ_ℓ using $\vee$. E.g., solving the equation of Ex. 6, among other equations that were omitted, results in a solution that includes, among others, the MHF states of Ex. 4. In what follows we assume that $\mathcal{H}_P$ has been solved, and let Φ_ℓ be the MHF state at ℓ in such solution.

Theorem 1. *For any program point $\ell \in \mathtt{ppoints}(P)$, we have $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$.*

In the rest of this article we let $\mathtt{mhf}_\alpha(\ell) = \{x \mid x \in V_m,\ \Phi_\ell \models x\}$, i.e., the set of finished future variables at ℓ that is induced by Φ_ℓ. Theorem 1 implies $\mathtt{mhf}_\alpha(\ell) \subseteq \mathtt{mhf}(\ell)$. Computing $\mathtt{mhf}_\alpha(\ell)$ using the MHF states of Ex. 4, among others that are omitted, results exactly in the MHF sets of Ex. 2.

5 MHP Analysis

In this section we present our MHP analysis, which is based on incorporating the MHF sets of Sec. 4 into the MHP analysis of [3]. In sections 5.1 and 5.2 we describe how we modify the two phases of the original analysis, and describe the gain of precision with respect to [3] in each phase.

5.1 Local MHP

The local MHP analysis (LMHP) considers each method m separately, and for each $\ell \in \texttt{ppoints}(m)$ it infers an LMHP state that describes the tasks that might be executing when reaching ℓ (considering only tasks invoked in m). An LMHP state Ψ is a *multiset* of MHP atoms, where each atom represents a task and can be: (1) $y{:}\ell'{:}\tilde{m}(\bar{x})$, which represents an *active* task that might be at any of its program points, including the exit one, and is bound to future variable y. Moreover, this task is an instance of method m that was called at program point ℓ' (the *calling site*) with future parameters $\bar{x}$; or (2) $y{:}\ell'{:}\hat{m}(\bar{x})$, which differs from the previous one in that the task can only be at the exit program point, i.e., it is a *finished* task. In both cases, future variables y and $\bar{x}$ can be $\star$, which is a special symbol indicating that we have no information on the future variable.

Intuitively, the MHP atoms of Ψ represent (local) tasks that are executing in parallel. However, since a variable y cannot be bound to more than one task at the same time, atoms bound to the same variable represent mutually exclusive tasks, i.e., cannot be executing at the same time. The same holds for atoms that use *mutually exclusive calling sites* ℓ_1 and ℓ_2 (i.e., there is no path from ℓ_1 and ℓ_2 and vice versa). The use of multisets allows including the same atom several times to represent different instances of the same method. We let $(a, i) \in \Psi$ indicate that a appears i times in Ψ. Note that i can be ∞, which happens when the atom corresponds to a calling site inside a loop, this guarantees convergence of the analysis. Recall that the MHP atoms of [3] do not use the parameters $\bar{x}$ and the calling site ℓ', since they do not benefit from such extra information.

Example 7. The following are LMHP states for some program points from Fig. 2:

L5 : $\{\mathsf{x{:}2{:}\tilde{f}(),z{:}3{:}\tilde{q}()}\}$
L7 : $\{\mathsf{x{:}2{:}\tilde{f}(),z{:}3{:}\tilde{q}(),w{:}6{:}\tilde{g}(x)}\}$
L10: $\{\mathsf{x{:}2{:}\tilde{f}(),z{:}3{:}\tilde{q}(),w{:}9{:}\tilde{k}(x,z)}\}$
L11: $\{\mathsf{x{:}2{:}\tilde{f}(),z{:}3{:}\tilde{q}(),w{:}6{:}\tilde{g}(x),w{:}9{:}\tilde{k}(x,z)}\}$
L12: $\{\mathsf{x{:}2{:}\hat{f}(),z{:}3{:}\tilde{q}(),w{:}6{:}\hat{g}(x),w{:}9{:}\hat{k}(x,z)}\}$
L16: $\{\mathsf{x{:}15{:}\tilde{f}()}\}$
L18: $\{\mathsf{x{:}15{:}\tilde{f}(),z{:}17{:}\tilde{g}(x)}\}$
L20: $\{\mathsf{x{:}15{:}\hat{f}(),z{:}17{:}\hat{g}(x)}\}$

L21: $\{\mathsf{x{:}15{:}\hat{f}(),z{:}17{:}\hat{g}(x)}\}$
L27: $\{\mathsf{z{:}26{:}\tilde{f}(),(\star{:}28{:}\hat{q}(),\infty),(\star{:}29{:}\hat{h}(\star,z),\infty)}\}$
L29: L27 $\cup$ $\{\mathsf{x{:}28{:}\tilde{q}()}\}$
L30: L29 $\cup$ $\{\mathsf{w{:}29{:}\tilde{h}(x,z)}\}$
L31: $\{\mathsf{z{:}26{:}\tilde{f}(),(\star{:}28{:}\hat{q}(),\infty),(\star{:}29{:}\hat{h}(\star,z),\infty)}\}$
L32: $\{\mathsf{z{:}26{:}\tilde{f}(),(\star{:}28{:}\hat{q}(),\infty),(\star{:}29{:}\hat{h}(\star,z),\infty)}\}$
L44: $\{\}$
L46: $\{\mathsf{z{:}45{:}\tilde{g}(a)}\}$
L48: $\{\mathsf{z{:}45{:}\hat{g}(a)}\}$

Let us explain some of the above LMHP states. The state at L5 includes $\mathsf{x{:}2{:}\tilde{f}()}$ and $\mathsf{z{:}3{:}\tilde{q}()}$ for the active tasks invoked at L2 and L3. The state at L11 includes an atom for each task invoked in m1. Note that those of g and h are bound to the same future variable w, which means that only one of them might be executing at L11, depending on which branch of the **if** statement is taken. The state at L12 includes z:3:$\hat{\mathsf{q}}$() since q might be active at L12 if we take the **then** branch of the **if** statement, and the other atoms correspond to tasks that are finished. The state at L27 includes z:26:$\tilde{\mathsf{f}}$() for the active task invoked at L26, and $\star$:28:$\hat{\mathsf{q}}$() and $\star$:29:$\hat{\mathsf{h}}$($\star$,z) with ∞ multiplicity for the tasks created inside the loop. Note that the first parameter of h is $\star$ since x is rewritten at each iteration.

The LMHP states are inferred by a *data-flow analysis* which is defined as a solution of a set of LMHP constraints obtained by applying the following transfer function τ to the instructions. Given an LMHP state Ψ_ℓ, the effect of executing instruction I_ℓ within Ψ_ℓ, denoted by $\tau(I_\ell)$, is defined as follows:

- if I_ℓ is a call $y = m(\bar{x})$, then $\tau(I_\ell) = \Psi_\ell[y/\star] \cup \{y{:}\ell'{:}\tilde{m}(\bar{x})\}$, which replaces each occurrence of y by $\star$, since it is rewritten, and then adds a new atom $y{:}\ell{:}\tilde{m}(\bar{x})$ for the newly created task. E.g., the LMHP state of L30 in Ex. 7 is obtained from the one of L29 by adding w:29:$\tilde{\mathsf{h}}$(x,z) for the call at L29;
- if I_ℓ is **await** y?, and ℓ' is the program point after ℓ, then we mark all tasks that are bound to a finished future variable as finished, i.e., $\tau(I_\ell)$ is obtained by turning each $z{:}\ell''{:}\tilde{m}(\bar{x}) \in \Psi_\ell$ to $z{:}\ell''{:}\hat{m}(\bar{x})$ for each $z \in \mathtt{mhf}_\alpha(\ell')$. E.g., the LMHP state of L12 in Ex. 7 is obtained from the one of L11 by turning the status of g, k, and f to finished (since w and x are finished at L12);
- otherwise, $\tau(I_\ell) = \Psi_\ell$.

The main difference w.r.t. the analysis of [3] is the treatment of **await** y?: while we use an MHF set computed using the inter-procedural MHF analysis of Sec. 4, in [3] the MHF set $\{y\}$ is used, which is obtained syntactically from the instruction. Our LMHP analysis, as [3], is defined as a solution of a set of LMHP constraints. In what follows we assume that the results of the LMHP analysis are available, and we will refer to the LMHP state of program point ℓ as Ψ_ℓ.

5.2 Global MHP

The results of the LMHP analysis are used to construct an MHP graph, from which we can compute the desired set of MHP pairs. The construction is exactly as in [3] except that we carry the new information in the MHP atoms. However, the process of extracting the MHP pairs from such graphs will be modified.

In what follows, we use $y{:}\ell{:}\check{m}(\bar{x})$ to refer to an MHP atom without specifying if it corresponds to an active or finished task, i.e., the symbol $\check{m}$ can be matched to $\tilde{m}$ or $\hat{m}$. As in [3], the nodes of the MHP graph consist of two method nodes $\tilde{m}$ and $\hat{m}$ for each method m, and a program point node ℓ for each $\ell \in \mathtt{ppoints}(P)$. Edges from $\tilde{m}$ to each $\ell \in \mathtt{ppoints}(m)$ indicate that when m is active, it can be executing at any program point, including the exit, but only one. An edge from $\hat{m}$ to ℓ_m indicates that when m is finished it can be only at its exit program point. The out-going edges from a program point node ℓ reflect the atoms of the LMHP state Ψ_ℓ as follows: if $(y{:}\ell'{:}\check{m}(\bar{x}), i) \in \Psi_\ell$, then there is an edge from node ℓ to node $\check{m}$ and it is labeled with $i{:}y{:}\ell'{:}\bar{x}$. These edges simply indicate which tasks might be executing in parallel when reaching ℓ, exactly as Ψ_ℓ does.

Example 8. The MHP graphs $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ in Fig. 2, correspond to methods m_1, m_2, and m_3, each analyzed together with its reachable methods. For simplicity, the graphs include only some program points of interest. Note that the out-going edges of program point nodes coincide with the LMHP states of Ex. 7.

The procedure of [3] for extracting the MHP pairs from the MHP graph of a program P, denoted $\mathcal{G}_P$, is based on the following principle: (ℓ_1, ℓ_2) is an MHP pair induced by $\mathcal{G}_P$ iff (i) $\ell_1 \rightsquigarrow \ell_2 \in \mathcal{G}_P$ or $\ell_2 \rightsquigarrow \ell_1 \in \mathcal{G}_P$; or (ii) there is a program point node ℓ_3 and paths $\ell_3 \rightsquigarrow \ell_1 \in \mathcal{G}_P$ and $\ell_3 \rightsquigarrow \ell_2 \in \mathcal{G}_P$, such that the first edges of these paths are different and they do not correspond to mutually exclusive MHP atoms, i.e., they use different future variables and do not correspond to mutually exclusive calling sites (see Sec. 5.1). Edges with multiplicity $i > 1$ represent i different edges. The first (resp. second) case is called direct (resp. indirect) MHP, see Sec. 3.

Example 9. Let us explain some of the MHP pairs induced by $\mathcal{G}_1$ of Fig. 2. Since $11 \rightsquigarrow 35 \in \mathcal{G}_1$ and $11 \rightsquigarrow 58 \in \mathcal{G}_1$, we conclude that (11,58) and (11,35) are direct MHP pairs. Moreover, since these paths originate in the same node 11, and the first edges use different future variables, we conclude that (58,35) is an indirect MHP pair. Similarly, since $11 \rightsquigarrow 38 \in \mathcal{G}_1$ and $11 \rightsquigarrow 50 \in \mathcal{G}_1$ we conclude that (11,38) and (11,50) are direct MHP pairs. However, in this case (38,50) is not an (indirect) MHP pair because the first edges of these paths use the same future variable w. Indeed, the calls to g and k appear in different branches of an **if** statement. To see the improvement w.r.t. to [3] note that node 12 does not have an edge to $\tilde{\mathsf{f}}$, since our MHF analysis infers that x is finished at that L12. The analysis of [3] would have an edge to $\tilde{\mathsf{f}}$ instead of $\hat{\mathsf{f}}$, and thus it produces spurious pairs such as (12,35). Similar improvements occur also in $\mathcal{G}_2$ and $\mathcal{G}_3$.

Now consider nodes 35 and 40, and note that we have $11 \rightsquigarrow 35 \in \mathcal{G}_1$ and $11 \rightsquigarrow 40 \in \mathcal{G}_1$, and moreover these paths use different future variables. Thus, we conclude that (35,40) is an indirect MHP pair. However, carefully looking at the program we can see that this is a spurious pair, because x (to which task f is bound) is passed to method g, as parameter w, and w is guaranteed to finish when executing **await** w? at L39. A similar behavior occurs also in $\mathcal{G}_2$ and $\mathcal{G}_3$. For example, the paths $30 \rightsquigarrow 58 \in \mathcal{G}_3$ and $30 \rightsquigarrow 40 \in \mathcal{G}_3$ induce the indirect MHP pair (58,40), which is spurious since x is passed to h at L29, as parameter a, which in turn is passed to g at L45, as parameter w, and w is guaranteed to finish when executing **await** w? at L39.

The spurious pairs in the above example show that even if we used our improved LMHP analysis when constructing the MHP graph, using the procedure of [3] to extract MHP pairs might produce spurious pairs. Next, we address this imprecision by modifying the process of extracting the MHP pairs to have an extra condition to eliminate such spurious MHP pairs. This condition is based on identifying, for a given path $\check{m} \rightsquigarrow \ell \in \mathcal{G}_P$, which of the parameters of m are guaranteed to finish before reaching ℓ, and thus, any task that is passed to m in those parameters cannot execute in parallel with ℓ.

Definition 2. *Let p be a path $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$, $\bar{z}$ be the formal parameter of m, and I a set of parameter indices of method m. We say that I is* not alive along p *if (i) p has a single edge, and for some $i \in I$ the parameter z_i is in $\mathtt{mhf}_\alpha(\ell)$; or (ii) p is of the form $\breve{m} \longrightarrow \ell_1 \xrightarrow{k:y:\ell':\bar{x}} \breve{m}_1 \rightsquigarrow \ell$, and for some $i \in I$ the parameter z_i is in $\mathtt{mhf}_\alpha(\ell_1)$ or $I' = \{j \mid i \in I, z_i = x_j\}$ is not alive along $\breve{m}_1 \rightsquigarrow \ell$.*

Intuitively, I is not alive along p if some parameter z_i, with $i \in I$, is finished at some point in p. Thus, any task bound to z_i cannot execute in parallel with ℓ.

Example 10. Consider $p \equiv \tilde{g} \rightsquigarrow 40 \in \mathcal{G}_1$, and let $I = \{1\}$, then I is not alive along p since it is a path that consists of a single edge and $\mathsf{w} \in \mathtt{mhf}_\alpha(40)$. Now consider $\tilde{h} \rightsquigarrow 40 \in \mathcal{G}_3$, and let $I = \{1\}$, then I is not alive along p since $I' = \{1\}$ is not alive along $\tilde{g} \rightsquigarrow 40$.

The notion of "*not alive along a path*" can be used to eliminate spurious MHP pairs as follows. Consider two paths

$$p_1 \equiv \ell_3 \xrightarrow{i_1:y_1:\ell_1':\bar{w}} \tilde{m}_1 \rightsquigarrow \ell_1 \in \mathcal{G}_P \quad \text{and} \quad p_2 \equiv \ell_3 \xrightarrow{i_2:y_2:\ell_2':\bar{x}} \breve{m}_2 \rightsquigarrow \ell_2 \in \mathcal{G}_P$$

such that $y_1 \neq \star$, and the first node after $\tilde{m}_1$ does not correspond to the exit program point of m_1, i.e., m_1 might be executing and bound to y_1. Define

- $F = \{y_1\} \cup \{y \mid \Phi_{\ell_3} \models y \rightarrow y_1\}$, i.e., the set of future variables at ℓ_3 such that when any of them is finished, y_1 is finished as well; and
- $I = \{i \mid y \in F,\ x_i = y\}$, i.e., the indices of the parameters of m_2 to which we pass variables from F (in p_2).

We claim that if I is not alive along p_2, then the MHP pair (ℓ_1, ℓ_2) is spurious. This is because before reaching ℓ_2, some task from F is guaranteed to terminate, and hence the one bound to y_1, which contradicts the assumption that m_1 is not finished. In such case p_1 and p_2 are called mutually exclusive paths.

Example 11. We reconsider the spurious indirect MHP pairs of Ex. 9. Consider first (35,40), which originates from

$$p_1 \equiv 11 \xrightarrow{1:\mathsf{x}:2:[]} \tilde{f} \rightsquigarrow 35 \in \mathcal{G}_1 \text{ and } p_2 \equiv 11 \xrightarrow{1:\mathsf{w}:6:[\mathsf{x}]} \tilde{g} \rightsquigarrow 40.$$

We have $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, and we have seen in Ex. 10 that I is not alive along $\tilde{g} \rightsquigarrow 40 \in \mathcal{G}_1$, thus p_1 and p_2 are mutually exclusive and we eliminate this pair. Similarly, consider (58,40) which originates from

$$p_1 \equiv 30 \xrightarrow{1:\mathsf{x}:28:[]} \tilde{q} \rightsquigarrow 58 \in \mathcal{G}_3 \text{ and } p_2 \equiv 30 \xrightarrow{1:\mathsf{w}:29:[\mathsf{x},\mathsf{z}]} \tilde{h} \rightsquigarrow 40.$$

Again $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, and we have seen in Ex. 10 that I is not alive along $\tilde{h} \rightsquigarrow 40 \in \mathcal{G}_3$, thus p_1 and p_2 are mutually exclusive and we eliminate this pair.

Recall that $\mathcal{E}_P$ is the set of all concrete MHP pairs. Let $\tilde{\mathcal{E}}_P$ be the set of all MHP pairs obtained by applying the process of [3], modified to eliminate indirect pairs that correspond to mutually exclusive paths.

Theorem 2. *$\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.*

6 Conclusions, Implementation and Related Work

The main contribution of this work has been the enhancement of an MHP analysis that could only handle a restricted form of intra-procedural synchronization to the more general inter-procedural setting, as available in today's concurrent languages. Our analysis has a wide application scope on the inference of the main properties of concurrent programs, namely the new MHP relations are essential to infer (among others) the properties of the termination, resource usage and deadlock freedom of programs that use inter-procedural synchronization.

The analysis has been implemented in SACO [2], a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects, which is able to infer deadlock, termination and resource boundedness of ABS programs [10] that follow the concurrent objects paradigm. Concurrent objects are based on the notion of concurrently running objects, similar to the actor-based and active-objects approaches [12, 13]. These models take advantage of the concurrency implicit in the notion of object to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. Concurrent objects communicate via *asynchronous* method calls and use **await** instructions to synchronize with the termination of the asynchronous tasks. Therefore, the abstract model used in Sec. 2 fully captures the MHP relations arising in ABS programs.

The implementation has been built on top of the original MHP analysis in SACO. The MHF analysis has been implemented and its output has been used within the local and global phases of the MHP analysis, which have been adapted to this new input as described in the technical sections. The remaining analyses in SACO did not require any modification and now they work for inter-procedural synchronization as well. Our method can be tried online at: `http://costa.ls.fi.upm.es/saco/web` by enabling the option `Inter-Procedural Synchronization` of the `MHP` analysis in the `Settings` section. One can then apply the MHP analysis by selecting it from the menu for the types of analyses and then clicking on `Apply`. All examples used in the paper are available in the folder `SAS15` adapted to the syntax of the ABS language. In the near future, we plan to apply our analysis to industrial case studies that are being developed in ABS but that are not ready for experimentation yet.

There is an increasing interest in asynchronous programming and in concurrent objects, and in the development of program analyses that reason on safety and liveness properties [6]. Existing MHP analyses for asynchronous programs [3, 11, 1] lose all information when future variables are used as parameters, as they do not handle inter-procedural synchronization. As a consequence, existing analysis for more advanced properties [9, 4] that rely on the MHP relations do all lose the associated analysis information on such futures. In future work we plan to study the complexity of our analysis, which we conjuncture to be in the same complexity order as [3]. In addition, we plan to study the computational complexity of deciding MHP, for our abstract models, with and without inter-procedural synchronizations in a similar way to what has been done in [5] for the problem of state reachability.

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. A. Bouajjani and M. Emmi. Analysis of Recursively Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(3):10, 2013.
6. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable Refinement Checking for Concurrent Objects. In *Proc. of POPL 2015*, pages 651–662. ACM, 2015.
7. 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.
8. C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimization. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.
9. A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS, pages 273–288. Springer, 2013.
10. 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.
11. 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.
12. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, LNCS, pages 275–299. Springer, 2010.
13. S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008.