Analysis of May-Happen-in-Parallel in Concurrent Objects

Elvira Albert, Antonio Flores-Montoya, Samir Genaim

Analysis of May-Happen-in-Parallel in Concurrent Objects*

Elvira Albert, Antonio Flores-Montoya, and Samir Genaim

Complutense University of Madrid, Spain

Abstract. This paper presents a *may-happen-in-parallel* (MHP) analysis for OO languages based on *concurrent objects*. In this concurrency model, objects are the concurrency *units* such that, when a method is invoked on an object o_2 from a task executing on object o_1, statements of the current task in o_1 may run in parallel with those of the (asynchronous) call on o_2, and with those of transitively invoked methods. The goal of the MHP analysis is to identify pairs of statements in the program that may run in parallel in any execution. Our MHP analysis is formalized as a method-level (*local*) analysis whose information can be modularly composed to obtain application-level (*global*) information.

1 Introduction

The actor-based paradigm [2] on which concurrent objects are based has lately regained attention as a promising solution to concurrency in OO languages. For many application areas, standard mechanisms like threads and locks are too low-level and have been shown to be error-prone and, more importantly, not *modular* enough. The concurrent objects model is based on considering objects as the concurrency units i.e., each object conceptually has a dedicated processor. Communication is based on asynchronous method calls with standard objects as targets. An essential feature of this paradigm is that task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. Data-driven synchronization is possible by means of so-called *future* variables [7] as follows. Consider an asynchronous method call `m` on object `o`, written as `f=o.m()`. Here, the variable `f` is a future which allows synchronizing with the result of executing task `m`. In particular, the instruction **await** `f?` allows checking whether `m` has finished, and lets the current task release the processor to allow another available task to take it.

* This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

This paper develops a *may-happen-in-parallel* (MHP) analysis for concurrent objects. The goal of an MHP analysis is to identify pairs of statements that can execute in parallel (see, e.g., [10]). In the context of concurrent objects, an asynchronous method invocation $\mathsf{f}{=}\mathsf{o}_2.\mathsf{m}()$; within a task t_1 executing in an object o_1 implies that the subsequent instructions of t_1 in o_1 may execute in parallel with the instructions of m within o_2. However, if the asynchronous call is synchronized with an instruction **await** f?, after executing such an *await*, it is ensured that the execution of the call to m has terminated and hence, the instructions after the **await** cannot execute in parallel with those of m. Inferring precise MHP information is challenging because, not only does the current task execute in parallel with m, but also with other tasks that are *transitively* invoked from m. Besides, two tasks can execute in parallel even if they do not have a transitive invocation relation. For instance, if we add an instruction $\mathsf{f}_3{=}\mathsf{o}_3.\mathsf{p}()$; below the previous asynchronous invocation to m in t_1, then instructions in p may run in parallel with those of m. This is a form of *indirect* MHP relation in which tasks run in parallel because they have a common ancestor. The challenge is to precisely capture in the analysis all possible forms of MHP relations.

It is widely recognized that MHP is an analysis of utmost importance [10] to understand the behaviour and verify the soundness of concurrent programs. On one hand, it is a basic analysis to later construct different kinds of verification and testing tools which build on it in order to infer more complex properties. For example, in order to prove termination (or infer the cost) of a simple loop of the form **while** (l!=**null**) {f=o.process(l.data); **await** f?; l=l.next;}, assuming l is a shared variable (i.e., field), we need to know the tasks that can run in parallel with the body of the loop to check whether the length of the list l can be modified during the execution of the loop by some other task when the processor is released (at the **await**). For concurrent languages which are not data-race free, MHP is fundamental in order to verify the absence of data-races. On the other hand, it provides very useful information to automatically extract the maximal level of parallelism for a program and improve performance. In the context of concurrent objects, when the methods running on two different objects may run in parallel, it can be profitable to deploy such objects on different machines in order to improve the overall performance. As another application, the programmer can use the results of the MHP analysis to identify bugs in the program related to fragments of code which should not run in parallel, but where the analysis spots possible parallel execution.

This paper proposes a novel MHP analysis for concurrent objects. The analysis has two main phases: we first infer *method-level* MHP information by locally analyzing each method and ignoring transitive calls. This local analysis, among other things, collects the *escape* points of method calls, i.e., those program points in which the asynchronous calls terminate but there might be transitive asynchronous calls not finished. In the next step, we modularly compose the method-level information in order to obtain *application-level* (*global*) MHP information. The composition is achieved by constructing an *MHP analysis graph* which over-approximates the parallelism –both implicit and through transitive calls– in the

application. Then, the problem of inferring if two statements x and y can run in parallel amounts to checking certain *reachability* conditions between x and y in the MHP analysis graph. We have implemented our analysis in COSTABS [3], a cost and termination analyzer for the ABS language. ABS [8] is an actor-like language which has been recently proposed to model distributed concurrent objects. The implementation has been evaluated on small applications which are classical examples of concurrent programming and on two industrial case studies. Results on the efficiency and accuracy of the analysis, in spite of being still prototypical, are promising.

2 Concurrent Objects

We describe the syntax and semantics of the simple imperative language with concurrent objects on which we develop our analysis. It is basically the subset of the ABS language [8] relevant to the MHP analysis. Class, method, field, and variable names are taken from a set $\mathcal{X}$ of valid *identifiers*. A *program* consists of a set of classes $\mathcal{K} \subseteq \mathcal{X}$. The set *Types* is the set of possible types $\mathcal{K} \cup \{\textsf{int}\}$, and the set $\mathit{Types}_{\mathcal{F}}$ is the set of future variable types defined as $\{Fut\langle t\rangle \mid t \in \mathit{Types}\}$. A *class declaration* takes the form:

$$\textsf{class}\ \ \kappa_1\ \{t_1\ fn_1; \ldots\ t_n\ fn_n;\ M_1\ \ldots\ M_k\}$$

where each "$t_i\ fn_i$" declares a field fn_i of type $t_i \in \mathit{Types}$, and each M_i is a method definition. A *method definition* takes the form

$$t\ m(t_1\ w_1, \ldots, t_n\ w_n)\ \{t_{n+1}\ w_{n+1}; \ldots t_{n+p}\ w_{n+p};\ s\}$$

where $t \in \mathit{Types}$ is the type of the return value; $w_1, \ldots, w_n \in \mathcal{X}$ are the formal parameters of types $t_1, \ldots, t_n \in \mathit{Types}$; $w_{n+1}, \ldots, w_{n+p} \in \mathcal{X}$ are local variables of types $t_{n+1}, \ldots, t_{n+p} \in \mathit{Types} \cup \mathit{Types}_{\mathcal{F}}$; and s is a sequence of instructions which adhere to the following grammar:

$$\begin{array}{rl}
e & ::= \textsf{null} \mid \textsf{this}.f \mid x \mid n \mid e + e \mid e * e \mid e - e \\
b & ::= e > e \mid e = e \mid b \wedge b \mid b \vee b \mid !b \\
s & ::= \mathit{instr} \mid \mathit{instr}; s \\
\mathit{instr} & ::= x{=}\textsf{new}\ \kappa(\bar{x}) \mid x{=}e \mid \textsf{this}.f{=}e \mid y{=}x.m(\bar{z}) \mid \textsf{return}\ x \\
& \quad \textsf{if}\ b\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{while}\ b\ \textsf{do}\ s \mid \textsf{await}\ y? \mid x{=}y.\textsf{get}
\end{array}$$

There is an implicit local variable called **this** that refers to the current object. x and y represent variables of types $t \in \mathit{Types}$ and $ft \in \mathit{Types}_{\mathcal{F}}$ respectively. Observe that only fields of the current object **this** can be accessed (this, together with the semantics, make the language be data-race free [8]). We assume the program includes a method called main without parameters, which does not belong to any class and has no fields, from which the execution will start. Data synchronization is by means of future variables as follows. An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}x.m(\bar{z})$ such that the **await** $y?$ is executed only when the future variable y is available (i.e., the

$$(1)\quad \frac{(O', l', s') = eval(instr, O, l, oid) \quad instr \in \{x{=}e,\ \textbf{this}.fn{=}e,\ x{=}\textbf{new}\ \kappa(\bar{x}),\ \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2,\ \textbf{while}\ b\ \textbf{do}\ s_3\}}{\langle O, \{\langle tid, m, oid, \top, l, instr; s\rangle \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \top, l', s'; s\rangle \parallel T\}\rangle}$$

$$(2)\quad \frac{l(x) = oid_1 \neq \textbf{null}, l' = l[y \rightarrow tid_1], l_1 = buildLocals(\bar{x}, m)), tid_1 \text{is a fresh id}}{\begin{array}{c}\langle O, \{\langle tid, m, oid, \top, l, y{=}x.m_1(\bar{x}); s\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid, m, oid, \top, l', s\rangle, \langle tid_1, m_1, oid_1, \bot, l_1, body(m_1)\rangle \parallel T\}\rangle\end{array}}$$

$$(3)\quad \frac{\langle oid, \bot, f\rangle \in O, O' = O[\langle oid, \bot, f\rangle / \langle oid, \top, f\rangle], v = l(x)}{\langle O, \{\langle tid, m, oid, \top, l, \textbf{return}\ x\rangle\} \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \bot, l, \epsilon(v)\rangle \parallel T\}\rangle}$$

$$(4)\quad \frac{l_1(y) = tid_2}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, \textbf{await}\ y?; s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle\end{array}}$$

$$(5)\quad \frac{}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, lk, l_1, \textbf{await}\ y?; s_1\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid_1, m_1, oid_1, lk, l_1, \textbf{release}; \textbf{await}\ y?; s_1\rangle \parallel T\}\rangle\end{array}}$$

$$(6)\quad \frac{\langle oid, \bot, f\rangle \in O, O' = O[\langle oid, \bot, f\rangle / \langle oid, \top, f\rangle]}{\langle O, \{\langle tid, m, oid, \top, l, \textbf{release}; s\rangle \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \bot, l, s\rangle \parallel T\}\rangle}$$

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

$$(8)\quad \frac{l_1(y) = tid_2, l_1' = l_1[x \rightarrow v]}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, x{=}y.\textbf{get}; s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle \rightsquigarrow \\ \langle O, \{\langle tid_1, m_1, oid_1, \top, l_1', s_1\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v)\rangle \parallel T\}\rangle\end{array}}$$

Fig. 1. Summarized semantics

task is finished). In the meantime, the processor can be released and some other pending task on this object can take it. In contrast, the instruction y.**get** unconditionally blocks the processor (no other task of the same object can run) until y is available, i.e., the execution of $m(\bar{z})$ on x is finished. Note that class fields and methods parameters cannot have future types, i.e, future variables are defined locally in each method and cannot be passed over. This is a restriction of the approach, however, programs that pass futures over can still be analyzed with some loss of precision by ignoring the non-local future variables.

W.l.o.g, we assume that all methods in the program have different names. As notation, we use $body(m)$ for the sequence of instructions defining method m, $P_{\mathcal{M}}$ for the set of method names defined in a program P, $P_{\mathcal{F}}$ for the set of future variable names defined in a program P.

2.1 Operational Semantics

A program state S is a tuple $S = \langle O, T \rangle$ where O is a set of objects and T is a set of tasks. Only one task can be *active* (running) in each object and has the object's *lock*. All other tasks are *pending* to be executed or *finished* if they terminated and released the lock. The set of objects O includes all available objects. An object takes the form $\langle oid, lk, f \rangle$ where oid is a unique identifier taken from an infinite set of identifiers $\mathcal{O}$, $lk \in \{\top, \bot\}$ indicates whether the object's lock is free ($\top$) or not ($\bot$), and $f : \mathcal{X} \to \mathcal{O} \cup \mathbb{Z} \cup \{\textsf{null}\}$ is a partial mapping from object fields to values. The set of tasks T represents those tasks that are being executed. Each task takes the form $\langle tid, m, oid, lk, l, s \rangle$ where tid is a unique identifier of the task taken from an infinite set of identifiers $\mathcal{T}$, m is the method name executing in the task, oid identifies the object to which the task belongs, $lk \in \{\top, \bot\}$ is a flag that indicates if the task has the object's lock or not, $l : \mathcal{X} \to \mathcal{O} \cup \mathcal{T} \cup \mathbb{Z} \cup \{\textsf{null}\}$ is a partial mapping from local (possibly future) variables to their values, and s is the sequence of instructions still to be executed. Given a task tid, we assume that $object(tid)$ returns the object identifier oid of the corresponding task. The execution of a program starts from the initial state $S_0 = \langle \{\langle 0, \bot, f \rangle\}, \{\langle 0, \textsf{main}, 0, \top, l, body(\textsf{main}) \rangle\} \rangle$ where f is an empty mapping (since main had no fields), and l maps local references and future variables to **null** and integer variables to 0.

The execution proceeds from S_0 by applying *non-deterministically* the semantic rules depicted in Fig. 1. We use the notation $\{t \parallel T\}$ to represent that task t is non-deterministically selected for execution. The operational semantics is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows: (1) executes an instruction in a task that has its object lock. These instructions may change the heap (global state), the local state and the sequence of instructions that are left to execute (in the case of an if-then-else or a while instruction). Such changes are captured in function *eval*. As the instructions executed in this rule are standard, they are summarized. (2) A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier which is associated to the corresponding future variable. (3) When **return** is executed, the return value is stored in v so that it can be obtained by the future variables that point to that task. Besides, the lock is released and will never be taken again by that task (the notation $O[o/o']$ is used to replace o by o' in O). Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$), though it does not disappear as other tasks might need to access its return value. (4) If the future variable we are awaiting for points to a finished task, the await can be completed. (5) The await can be substituted by a release plus an await. This allows us to await until rule (4) can be applied. (6) A task executes a release and yields the lock so that any other task of the same object can take it. (7) A non finished task can obtain its object lock if it is unlocked. (8) A y.**get** instruction waits for the future variable but without yielding the lock. It then retrieves the value associated with the future variable y.

A	B	C	D	E

1 int m() {
2 ...
3 y=x.p();
4 z=x.q();
5 ...
6 await z?;
7 ...
8 await y?;
9 ...
10 }
	11 int m() {
	12 ...
	13 y=this.r();
	14 z=x1.p();
	15 z=x2.p();
	16 z=x3.q();
	17 w=z.get;
	18 ...
	19 await y?;
	20 }
		21 int m() {
		22 ...
		23 while b do
		24 y=x.q();
		25 await y?;
		26 z=x.p();
		27 ...
		28 ...
		29 ...
		30 }
			31 int m() {
			32 ...
			33 if b then
			34 y=x.p();
			35 else
			36 y=x.q();
			37 ...
			38 await y?;
			39 ...
			40 }
				41 int p() {
				42 y=x.r();
				43 ...
				44 }
				45 int q() {
				46 y=x.r();
				47 await y?;
				48 ...
				49 ...
				50 }

Fig. 2. Simple examples for different MHP behaviours.

3 Definition of MHP

We first formally define the concrete property "MHP" that we want to approximate using static analysis. In what follows, we assume that instructions are labelled such that it is possible to obtain the corresponding program point identifiers. We also assume that program points are globally different. We use $p_{\hat{m}}$ to refer to the entry program point of method m, and $p_{\dot{m}}$ to all program points after its **return** instruction. The set of all program points of P is denoted by $P_{\mathcal{P}}$. We write $p \in m$ to indicate that program point p belongs to method m. Given a sequence of instructions s, we use $pp(s)$ to refer to the program point identifier associated with its first instruction, $pp(\epsilon(v)) = p_{\dot{m}}$ and and $pp(\textbf{release}; s) = pp(s)$.

Definition 1 (concrete MHP). *Given a program P, its MHP is defined as $\mathcal{E}_P = \cup\{\mathcal{E}_S | S_0 \rightsquigarrow^* S\}$ where for the state $S = \langle O, Tk\rangle$, the set $\mathcal{E}_S$ is $\mathcal{E}_S = \{(pp(s_1), pp(s_2)) \mid \langle id_1, m_1, o_1, lk_1, l_1, s_1\rangle \in Tk, \langle id_2, m_2, o_2, lk_2, l_2, s_2\rangle \in Tk, id_1 \neq id_2\}$.*

Observe in the above definition that, as execution is non-deterministic (and different MHP behaviours can actually occur using different task scheduling strategies), the union of the pairs obtained from all derivations from S_0 is considered.

Let us explain first the notions of *direct* and *indirect* MHP and *escaped* methods, which are implicit in the definition of MHP above, on the simple representative patterns in Fig. 2. There are 4 versions of m which use the methods p, q and r. We consider a call to m with no other processes executing. Only the parts of p and q useful for explaining the MHP behavior are shown (the code of r is irrelevant). We implicitly assume that the last instruction of each method is a **return**. The global MHP behavior of executing each main (separately) is as follows.

(A) p is called from m, then r is called from p and q. The await instruction in program point 6 (L6 for short) ensures that q will have finished afterwards. If q has finished executing, its call to r has to be finished as well because there is an await in L47. The await instruction in L8 waits until p has finished before

continuing. That means that at L9, p is not longer executing. However, the call to r from p might be still executing. We say that r might *escape* from p. Method calls that might escape need to be considered.

(B) In example B, both q and p are called from m, but p is called twice. Any program point of p, for example L43, might execute in parallel with q even if they do not call each other, i.e., they have an *indirect* MHP relation. Furthermore, L43 might execute in parallel with any point of m after the method call, L15 − 17. We say that m is a common *ancestor* of p and q. Two methods execute indirectly in parallel if they have a common ancestor. Note that m is also a common ancestor of the two instances of p, so p might execute in parallel with itself. r is called in L13. However, as r belongs to the same object as m, it will not be able to start executing until m reaches a release point (L19). We say that r is *pending* from L14 up to L19.

(C) In the third example we have a **while** loop. If we do not estimate the number of iterations, we can only assume that q and p are called an arbitrary number of times. However, as every call to q has a corresponding await, q will not execute in parallel with itself. At L28, we might have any number of p instances executing but none of q. Note that if any method escaped from q, it could also be executing at L28.

(D) The last example illustrates an **if** statement. Either p or q is executed but not both. At L37, p or q might be executing but p and q cannot run in parallel even if m is a common ancestor. Furthermore, after the await instruction (L38) neither q or p might be executing. This information will be extracted from the fact that both calls use the same future variable.

4 MHP Analysis

The problem of inferring $\mathcal{E}_P$ is clearly undecidable in our setting [9], and thus we develop a MHP analysis which statically approximates $\mathcal{E}_P$. The analysis is done in two main steps, first it infers method-level MHP information. Then, in order to obtain application-level MHP, it composes this information by building a MHP graph whose paths provide the required global MHP information.

4.1 Inference of method-level MHP

The method-level MHP analysis is used to infer the local effect of each method on the global MHP property. In particular, for each method m, it infers, for each program point $p \in m$, the status of all tasks that (might) have been invoked (within m) so far. The status of a task can be (1) *pending*, which means that it has been invoked but has not started to execute yet, i.e., it is at the entry program point; (2) *finished*, which means that it has finished executing already, i.e., it is at the exit program point; and (3) *active*, which means that it can be executing at any program point (including the entry and the exit). As we explain later, the distinction between these statuses is essential for precision.

The analysis of each method abstractly executes its code such that the (abstract) state at each program point is a multiset of symbolic values that describes the status of all tasks invoked so far. Intuitively, when a method is invoked, we add it to the multiset (as pending or active depending if it is a call on the same object or on a different object); when an **await** y? or y.**get** instruction is executed, we change the status of the corresponding method to finished; and when the execution passes through a release point (namely **await** y? or **return**), we change the status of all pending methods to active.

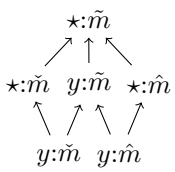
Example 1. Consider programs A and B in Fig. 2. The call to p (resp. q) at L3 (resp. L4) creates an *active* task that becomes *finished* at L8 (resp. L6). In B, the call to r at L13 creates a *pending* task that becomes *active* at L19 and *finished* after L19. p is an *active* task from L14 up to the end of the method. p will never become a *finished* task as its associated future variable is reused in L16.

The symbolic values used to describe the status of a task, referred to as MHP *atoms*, can be one of the following: (1) $y{:}\tilde{m}$, which represents an *active* task that is an instance of method m; (2) $y{:}\hat{m}$, which represents a *finished* task that is an instance of method m; and (3) $y{:}\check{m}$, which represents a *pending* task that is an instance of method m. In the three cases the task is associated to the future variable y. In addition, since it is not always possible to relate tasks to future variables (e.g., if they are reused), we also allow symbolic values in which y is replaced by $\star$, i.e., $\star$ represents any future variable.

Intuitively, an abstract state M is a multiset of MHP atoms which represents the following information: each $y{:}x \in M$ (resp. $\star{:}x \in M$) represents *one* task that *might* be available and associated to future variable y (resp. to any future variable). The status of the task is active, pending or finished, resp., if $x = \tilde{m}$, $x = \check{m}$ or $x = \hat{m}$. In addition, we can have several tasks associated to the same future variable meaning that at most one of them can be available at the same time (since only one task can be associated to a future variable in the semantics).

Example 2. Consider programs A, B and D. The multisets $\{y{:}\tilde{p}, z{:}\tilde{q}\}$, $\{y{:}\tilde{p}, z{:}\hat{q}\}$, $\{y{:}\hat{p}, z{:}\hat{q}\}$, $\{y{:}\check{r}, z{:}\tilde{p}\}$, $\{y{:}\check{r}, \star{:}\tilde{p}, \star{:}\tilde{p}, z{:}\hat{q}\}$ and $\{y{:}\tilde{p}, y{:}\tilde{q}\}$ resp. describe the abstract states at L5, L7, L9, L15, L18 and L37. An important observation is that, in the multiset of L18, when the future variable is reused, its former association is lost (and hence becomes $\star$). However, multiple associations to one future variable can be kept when they correspond to disjunctive branches, as in L37.

For a given program P, the set of all MHP atoms $\mathcal{A} = \{y{:}x \mid m \in P_{\mathcal{M}}, x \in \{\tilde{m}, \hat{m}, \check{m}\}, y \in P_{\mathcal{F}} \cup \{\star\}\}$ is a partially order set w.r.t. the partial order relation $\preceq$ defined as in the diagram below (we use $\prec$ for strict inequality and $=$ for syntactic equality). The meaning of $a \preceq a'$ is that concrete scenarios described by a, are also described by a'. For example, $y{:}\check{m} \preceq y{:}\tilde{m}$ because $y{:}\check{m}$ is included in the description of $y{:}\tilde{m}$ since an active task can be at any program point (including the entry program point). The set of all multisets over $\mathcal{A}$ is denoted by $\mathcal{B}$. We write $(a, i) \in M$ to indicate that a appears exactly $i > 0$



$$
\begin{array}{lrll}
(1) & \tau(y{=}x.m(\bar{x}), M) = M[y{:}x/\star{:}x] \cup \{y{:}\tilde{m}\} & & x \in \{\check{m}, \tilde{m}, \hat{m}\} \\
(2) & \tau(y{=}\textbf{this}.m(\bar{x}), M) = M[y{:}x/\star{:}x] \cup \{y{:}\check{m}\} & & x \in \{\check{m}, \tilde{m}, \hat{m}\} \\
(3) & \tau(\textbf{await}\ y?, M) = \tau(x{=}y.\textbf{get}, M) = M[y{:}z/y{:}\hat{m}] & & z \in \{\check{m}, \tilde{m}\} \\
(4) & \tau(\textbf{release}, M) = \tau(\textbf{return}, M) = M[y{:}\check{m}/y{:}\tilde{m}] & & \forall y \\
(5) & \tau(b, M) = M & \text{otherwise} &
\end{array}
$$

Fig. 3. Method-level MHP transfer function: $\tau : s \times \mathcal{B} \mapsto \mathcal{B}$.

times in M. In the examples, we omit i when it is 1. Given $M_1, M_2 \in \mathcal{B}$, we say that $a \in M_2$ *covers* $a' \in M_1$ if $a' \preceq a$. Thus, $M_1 \sqsubseteq M_2$ if all elements of M_1 are covered by *different* elements from M_2.

Note that for two different $M_1, M_2 \in \mathcal{B}$, it might be the case that $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$, in such case they represent the same concrete states. This happens because when $(a, \infty) \in M$, then any $(a', i) \in M$ is redundant if $a' \preceq a$. The join (or upper bound) of M_1 and M_2, denoted $M_1 \sqcup M_2$, is an operation that calculates a multiset $M_3 \in \mathcal{B}$ such that $M_1 \sqsubseteq M_3$ and $M_2 \sqsubseteq M_3$. It is not guaranteed that least upper bound exists, as we show in the following example.

Example 3. Let $M_1 = \{y{:}\hat{m}, y{:}\check{m}\}$ and $M_2 = \{y{:}\tilde{m}\}$. Both $M_3 = \{y{:}\hat{m}, y{:}\tilde{m}\}$ and $M_3' = \{y{:}\check{m}, y{:}\tilde{m}\}$ are upper bounds for M_1 and M_2. However, there is no other upper bound M_3'' such that $M_3'' \sqsubseteq M_3$ and $M_3'' \sqsubseteq M_3'$. Thus, the least upper bound of M_1 and M_2 does not exist.

The above example shows that there are several possible ways of computing an upper bound M_3 of two given abstract states M_1 and M_2. Assuming that multisets are normalized in the sense that redundant elements are removed, the following steps define a possible algorithm:

1. any atom in M_1 (resp. M_2) with ∞ multiplicity is added to M_3 and removed from M_1 (resp. M_2);
2. the atoms of M_1 or M_2 that are covered by an element of M_3 with infinite multiplicity are removed;
3. the atoms $M_1 \cap M_2$ are added to M_3, and removed from M_1 and M_2;
4. let $a \in M_1$ be an atom covered by an atom $a' \in M_2$ ($a \preceq a'$). Both a and a' are removed from M_1 and M_2 and a' is added to M_3. Respectively, if $a' \preceq a$, a is the one added to M_3. Note there are several possible ways to compute the covering as we have seen in the example above; and
5. $M_1 \cup M_2$ is added to M_3.

In what follows, for the sake of simplicity, we assume that the program to be analyzed has been instrumented to have a **release** instruction before every **await** y?. This is required to simulate the auxiliary instruction **release** introduced in the semantics described in Sec. 2.1 (we could simulate it implicitly in the analysis also). The analysis of a program P is done as follows. For each method $m \in P_{\mathcal{M}}$, it starts from an abstract state $\emptyset \in \mathcal{B}$, which assumes that there are no tasks

executing (since we are looking at the locally invoked tasks), and propagates the information to the different program points by applying the transfer function τ defined in Fig. 3 on the code $body(m)$. The transfer function defines the effect of executing each (simple) instruction on a given abstract state $M \in \mathcal{B}$. Let us explain the different cases of τ: Case 1 adds an active instance of m to the abstract state; Case 2 adds a pending instance of m to the abstract state; Case 3 changes the status all active tasks that are guaranteed to be finished; Case 4 changes all pending tasks to active tasks; and Case 5 applies to the remaining instructions which do not have any effect on the MHP information.

Example 4. Consider program B. The abstract state at L13 is $\emptyset$ since we have not invoked any method yet. Executing L13 adds $y{:}\check{r}$ since the call is to a method in the same object; executing L14 adds $z{:}\tilde{p}$; executing L16 renames one $z{:}\tilde{p}$ to $\star{:}\tilde{p}$ since the future variable z is reused, and adds $z{:}\tilde{q}$; executing L17 renames $z{:}\tilde{q}$ to $z{:}\hat{q}$ since it is guaranteed that q has finished. The auxiliary **release** between L18 and L19 renames $y{:}\check{r}$ to $y{:}\tilde{r}$, since the current task might suspend and thus any pending task might become active. Finally, L19 renames $y{:}\tilde{r}$ to $y{:}\hat{r}$.

The analysis merges abstract states at branching points (i.e., after **if** and at loop entries) using the join operation $\sqcup$. The analysis of while loops requires iterating the corresponding code several times until a fixpoint is reached. To guarantee convergence in such cases we employ the following widening operator $\triangle : \mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ after some predetermined number of iterations. Briefly, assuming that M_2 is the current abstract state at the loop entry program point, and that $M_1 \sqsubseteq M_2$ is the abstract state at the previous iteration, then $M_1 \triangle M_2$ replaces each element $(a, i) \in M_2$ by (a, ∞) if $(a, j) \in M_1$ and $i > j$, i.e., it replaces unstable elements by infinite number of occurrences in order to stabilize them.

Example 5. Let us demonstrate the analysis of the **if** and **while** statements on programs C and D. (**if**) At L37, the information that comes from the **then** and **else** branches is joined using $\sqcup$, namely $\{y{:}\tilde{p}\} \sqcup \{y{:}\tilde{q}\} = \{y{:}\tilde{p}, y{:}\tilde{q}\}$. Note that this state describes that either `q` or `p` are running at L37, but not both (as they share the same future variable); (**while**) In the first visit to L23, we have the abstract state $M_0 = \emptyset$, abstractly executing the body we reach L23 again with $M_1 = \{y{:}\hat{q}, z{:}\tilde{p}\}$ and joining it with M_0 results in M_1 itself. Similarly, if we apply two more iterations we respectively get $M_2 = \{\star{:}\hat{q}, \star{:}\tilde{p}, y{:}\hat{q}, z{:}\tilde{p}\}$ and $M_3 = \{(\star{:}\hat{q}, 2), (\star{:}\tilde{p}, 2), y{:}\hat{q}, z{:}\tilde{p}\}$. Inspecting M_2 and M_3, we see that $\star{:}\hat{q}$ and $\star{:}\tilde{p}$ are unstable, thus, we apply the widening operator $M_2 \triangle M_3$ obtaining $M_3' = \{(\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty), y{:}\hat{q}, z{:}\tilde{p}\}$. Executing the loop body starting with the new abstract state does not add any new MHP atoms since $\star{:}\hat{q}$ and $\star{:}\hat{p}$ already appear an infinite number of times.

In what follows, we assume that the result of the analysis is a mapping $\mathcal{L}_{\mathcal{P}}{:}P_{\mathcal{P}} \mapsto \mathcal{B}$ from each program point p (including entry and exit points) to an abstract state $\mathcal{L}_{\mathcal{P}}(p) \in \mathcal{B}$ that describes the status of the tasks that might be executing at p.

Example 6. The following table summarizes $\mathcal{L}_{\mathcal{P}}$ for some selected program points of interest (from Fig. 2) that we will use in the next section:

4:$\{y{:}\tilde{p}\}$	16:$\{y{:}\check{r}, z{:}\tilde{p}, \star{:}\tilde{p}\}$	25:$\{y{:}\tilde{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	44 :$\{y{:}\tilde{r}\}$
6:$\{y{:}\tilde{p}, z{:}\tilde{q}\}$	17:$\{y{:}\check{r}, (\star{:}\tilde{p}, 2), z{:}\tilde{q}\}$	26:$\{y{:}\hat{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	47 :$\{y{:}\tilde{r}\}$
8:$\{y{:}\tilde{p}, z{:}\hat{q}\}$	18:$\{y{:}\check{r}, (\star{:}\tilde{p}, 2), z{:}\hat{q}\}$	30:$\{y{:}\hat{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	50 :$\{y{:}\hat{r}\}$
10:$\{y{:}\hat{p}, z{:}\hat{q}\}$	20:$\{y{:}\hat{r}, (\star{:}\tilde{p}, 2), z{:}\hat{q}\}$	38:$\{y{:}\tilde{p}, y{:}\tilde{q}\}$	
14:$\{y{:}\check{r}\}$	24:$\{y{:}\hat{q}, (\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty)\}$	40:$\{y{:}\hat{p}, y{:}\hat{q}\}$	

Recall that the state associated to a program point represents the state before the execution of the corresponding instruction. In addition, the results for the entry points L2, L12, L22, L32, L42 and L46 are all $\emptyset$. Also note that L10, L20, L30, L40, L44 and L50 are exit points for the corresponding methods. Those will allow us to capture tasks that escape from the methods. Observe that L24, L26 and L30 contain redundant information because $y{:}\hat{q}$ is redundant w.r.t. $(\star{:}\hat{q}, \infty)$.

4.2 The Notion of MHP Graph

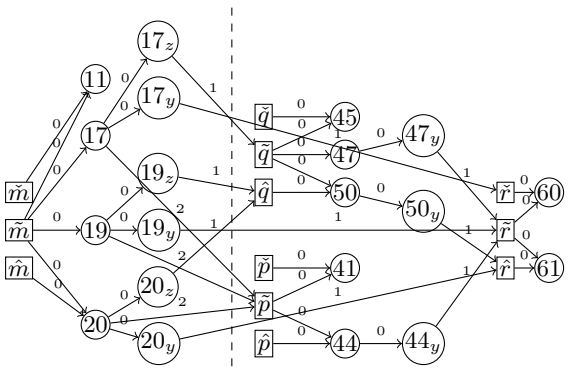
We now introduce the notion of *MHP graph* from which it is possible to extract precise information on which program points might globally run in parallel (according to Def. 1). A MHP graph has different types of nodes and different types of edges. There are nodes that represent the status of methods (active, pending or finished) and nodes which represent the program points. Outgoing edges from method nodes represent points of which at most one might be executing. In contrast, outgoing edges from program point nodes represent tasks such that any of them might be running. The information computed by the method-level MHP analysis is required to construct the MHP graph. When two nodes are directly connected by $i > 0$ edges, we connect them with a single edge of weight i. We start by formally constructing the MHP graph for a given program P, and then explain the construction in detail.

Definition 2 (MHP graph). *Given a program P, and its method-level MHP analysis result $\mathcal{L}_P$, the MHP graph of P is a directed graph $\mathcal{G}_P = \langle V, E\rangle$ with a set of nodes V and a set of edges $E = E_1 \cup E_2 \cup E_3$ defined as follows:*

$$
\begin{aligned}
V &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \cup \{p_y \mid p \in P_{\mathcal{P}}, y{:}m \in \mathcal{L}_P(p)\}\\
E_1 &= \{\tilde{m} \xrightarrow{0} p \mid m \in P_{\mathcal{M}}, p \in P_{\mathcal{P}}, p \in m\} \cup \{\hat{m} \xrightarrow{0} p_{\hat{m}}, \check{m} \xrightarrow{0} p_{\check{m}} \mid m \in P_{\mathcal{M}}\}\\
E_2 &= \{p \xrightarrow{i} x \mid p \in P_{\mathcal{P}}, (\star{:}x, i) \in \mathcal{L}_P(p)\}\\
E_3 &= \{p \xrightarrow{0} p_y, p_y \xrightarrow{1} x \mid p \in P_{\mathcal{P}}, (y{:}x, i) \in \mathcal{L}_P(p)\}
\end{aligned}
$$

Let us explain the different components of $\mathcal{G}_P$. The set of nodes V consists of several kinds of nodes:

1. *Method nodes*: Each $m \in P_{\mathcal{M}}$ contributes three nodes $\tilde{m}$, $\hat{m}$, and $\check{m}$. These nodes will be used to describe the program points that can be reached from active, finished or pending tasks which are instances of m.
2. *Program point nodes*: Each $p \in P_{\mathcal{P}}$ contributes a node p that will be used to describe which other program points might be running in parallel with it.
3. *Future variable nodes*: These nodes are a refinement of program point nodes for improving precision in the presence of branching constructs. Each future



	11	17	19	20	45	47	50	41	44	60	61
11											
17					●	●	●	●	●	●	●
19							●	●	●	●	●
20							●	●	●	●	●
45	●							○	○	○	○
47	●							○	○	●	●
50	●	●	●					○	○	○	●
41	●	●	●	○	○	○	○	○	○	○	○
44	●	●	●	○	○	○	○	○		●	●
60	●	●	●	○	●	○	○	●	○		○
61		●	●	●	○	●	●	○	●	○	○

Fig. 4. The $\mathcal{G}_{P}$ of example B (left) and its corresponding $\tilde{\mathcal{E}}_{P}$ (right).

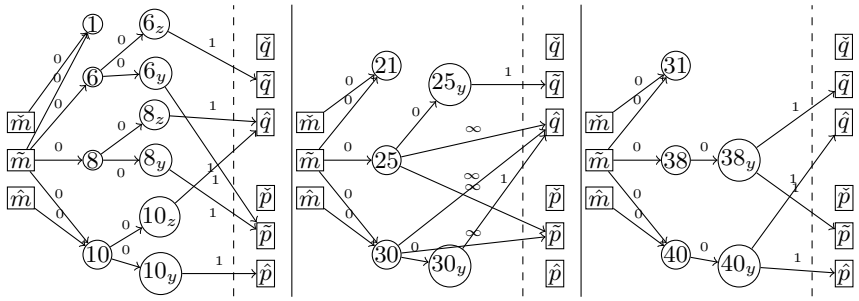


Fig. 5. Partial $\mathcal{G}_{P}$ for examples: A, C and D

variable y that appears in $\mathcal{L}_{P}(p)$ contributes a node p_y. These nodes will be used to state that if there are several MHP atoms in $\mathcal{L}_{P}(p)$ that are associated to y, then at most one of them can be running.

What gives the above meaning to the nodes are the edges $E = E_1 \cup E_2 \cup E_3$:

1. Edges in E_1 describe the program points at which each task can be depending on its status. Each m contributes the edges (a) $\tilde{m} \xrightarrow{0} p$ for each $p{\in}m$, which means that if m is active it can be in a state in which any of its program points is executing (but only one of them); (b) $\check{m} \xrightarrow{0} p_{\check{m}}$, which means that when m is pending, it is at the entry program point; and (c) $\hat{m} \xrightarrow{0} p_{\hat{m}}$, which means that when m is finished, it is at the exit program point;
2. Edges in E_2 describe which tasks might run in parallel with such program point. For every program point $p \in P_{\mathcal{P}}$, if $(\star{:}x, i) \in \mathcal{L}_{P}(p)$ then $p \xrightarrow{i} x$ is added to E_2. This edges means, if $x = \tilde{m}$ for example, that up to i instances of m might be running in parallel when reaching p;
3. Edges in E_3 enrich the information for each program point given in E_2. An edge $p_y \xrightarrow{1} x$ is added to E_3 if $(y{:}x, i) \in \mathcal{L}_{P}(p)$. For each future variable

y that appears in $\mathcal{L}_P(p)$ an edge $p \xrightarrow{0} p_y$ is also added to E_3. This allows us to accurately handle cases in which several MHP atoms in $\mathcal{L}_P(p)$ are associated to the same future variable. Recall that in such cases at most one of the corresponding tasks can be available (see Ex. 2).

Note that MHP graphs might have cycles due to recursion.

Example 7. Using the method-level MHP information of Ex. 6 we obtain the MHP graphs for the four examples. Fig. 4 contains (to the left) the graph of example B and Fig. 5 contains incomplete graphs of examples A, C and D. The omitted parts of the graphs for A, C and D, marked with dashed lines, should be identical to the subgraph to the right of the dashed line in the graph of B. Besides, for readability, the graphs do not include all program points, but rather only those that correspond to entry, get and release points.

4.3 Inference of Global MHP

Given the MHP graph $\mathcal{G}_P$, two program points $p_1, p_2 \in P_{\mathcal{P}}$ may run in parallel (i.e., it might be that $(p_1, p_2) \in \mathcal{E}_P$) if one of the following conditions hold:

1. there is a non-empty path in $\mathcal{G}_P$ from p_1 to p_2 or vice-versa; or
2. there is a program point $p_3 \in P_{\mathcal{P}}$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$.

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task running from which it is possible to *transitively* reach p_2, or vice-versa. This is the case, for example, of program points 17 and 50 in Fig. 4. The second case corresponds to *indirect MHP* scenarios in which a task is running at p_3 and there are two other tasks p_1 and p_2 executing in parallel and both are reachable from p_3. This is the case, for example, of program points 50 and 44 that are both reachable from program point 19 in Fig. 4 through paths that start with a different edge. Observe that the first edge can only be shared if it has weight $i > 1$ because it represents that there might be more than one instance of the same type of task running. This allows us to infer that 41 may run in parallel with itself because the edge from 17 to $\tilde{p}$ has weight 2 and, besides, that 41 can run in parallel with 44. Note that program points 45, 47, and 50 of method q do not satisfy any of the above conditions, which implies, as expected, that they cannot run in parallel.

The following definition formalizes the above intuition. We write $p_1 \rightsquigarrow p_2 \in \mathcal{G}_P$ to indicate that there is a path of length at least 1 from p_1 to p_2 in $\mathcal{G}_P$, and $p_1 \xrightarrow{i} x \rightsquigarrow p_2$ to indicate that such path starts with an edge to x with weight i.

Definition 3. *Given a program P, we let $\tilde{\mathcal{E}}_P = directMHP \cup indirectMHP$ where*

$$directMHP = \{(p_1, p_2) \mid p_1, p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P)\}$$
$$indirectMHP = \{(p_1, p_2) \mid p_1, p_2, p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P, x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\}$$

Example 8. The table on the right side of Fig. 4 represents the $\tilde{\mathcal{E}}_P$ obtained from the graph on the left side. Empty cells mean that the corresponding points cannot run in parallel. Cells marked by • indicate that the pair is in *directMHP*. Cells marked with ∘ indicate that the pair is in *indirectMHP*. Note that the table captures the MHP relations informally discussed in Sec. 3.

4.4 Soundness and Complexity

The following theorem states the soundness of the analysis, namely, that $\tilde{\mathcal{E}}_P$ is an over-approximation of $\mathcal{E}_P$.

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

As regards complexity, we distinguish its three phases:

1. The $\mathcal{L}_P$ computation can be performed independently for each method m. The transfer function τ only needs to be applied a constant number of times for each program point, even for loops, due to the use of widening. It is possible to represent multisets such that the cost of all multiset operations is linear w.r.t. their sizes which are at most $nm_m \cdot fut_m$, where nm_m is the number of different methods that can be called from m and fut_m is the number of future variables in m. Therefore, the cost of computing $\mathcal{L}_P$ for a method m is in $O(pp_m \cdot nm_m \cdot fut_m)$ where pp_m is the number of program points in the method.
2. The cost of creating the graph $\mathcal{G}_P$ is linear with respect to the number of edges. The number of edges originating from a method m is in $O(pp'_m \cdot nm_m \cdot fut_m)$ where pp'_m is the number of program points of interest. A strong feature of our analysis is that most of the program points can be ignored in this phase without affecting correctness or precision. Only points that correspond to **await** and **get** instructions and exit points are required for correctness. This happens due to the definition of τ in which the abstract states always grow (in the domain) except for those points.
3. Once the graph has been created, computing $\tilde{\mathcal{E}}_P$ is basically a graph reachability problem. Therefore, a straightforward algorithm for inferring of $\tilde{\mathcal{E}}_P$ is clearly in $O(n^3)$ where n is the number of nodes of the graph. However, a major advantage of this analysis is that for most applications there is no need to compute the complete $\tilde{\mathcal{E}}_P$; rather, this information can be obtained *on demand*.

5 Experimental Evaluation

We have implemented our analysis as a module of COSTABS [4], a cost analyzer of ABS programs. A standalone version of the MHP analysis can be tried out online at: http://costa.ls.fi.upm.es/costabs/mhp. Experimental evaluation has been carried out using two industrial case studies: ReplicationSystem and TradingSystem, which can be found at http://www.hats-project.eu, as well as a few

Code	Ns	$\mathbf{NP}_{\mathcal{P}}$	$\mathbf{E_p}$	$\tilde{\mathcal{E}}_{\mathbf{P}}$	$\mathbf{PPs^2}$	$\mathbf{R}\varepsilon$	$\mathbf{T_G}$	$\mathbf{T}_{\tilde{\mathcal{E}}_{\mathbf{P}}}$
RepSystem	496	213	-	7724	45369	-	360	23020
TradingSystem	360	137	-	14829	18769	-	120	18120
MailServer	23	8	17	34	64	26.5%	10	< 10
BookShop	35	21	66	66	196	0%	< 10	10
PeerToPeer	75	36	385	487	1296	7.87%	20	100
BBuffer	22	7	36	36	49	0%	< 10	< 10
Chat	120	45	552	1219	2025	32.9%	< 10	190
DistHT	51	24	83	151	573	11.8%	< 10	20

Table 1. Statistics about the analysis execution (times are in milliseconds)

typical concurrent applications: PeerToPeer, a peer to peer protocol implementation; Chat, a client-server implementation of a chat program; MailServer, a simple model of a Mail server; BookShop, a web shop client-server application; BBuffer, a classical bounded-buffer for communicating several producers and consumers; and DistHT, a distributed hash-table.

Table 1 summarizes our experiments. They have been performed on an Intel Core i5 at 2.4GHz with 3.7GB of RAM, running Linux. For each program, $\mathcal{G}_{\mathcal{P}}$ is built and the relation $\tilde{\mathcal{E}}_P$ is completely computed using only the program points required for soundness. **Ns** is the number of nodes of $\mathcal{G}_{\mathcal{P}}$ and $\mathbf{NP}_{\mathcal{P}}$ is the number of program point nodes. $\mathbf{E_p}$ is the number of MHP pairs obtained by running the program using a random scheduler, i.e., one which randomly chooses the next task to execute when the processor is released. These executions are bounded to a maximum number of interleavings as termination in some examples is not guaranteed. Observe that $\mathbf{E_p}$ does not capture all possible MHP pairs but just gives us an idea of the level of real parallelism. It gives us a lower bound of $\mathcal{E}_P$ which we will use to approximate the error. $\tilde{\mathcal{E}}_{\mathbf{P}}$ is the number of pairs inferred by the analysis. $\mathbf{PPs^2}$ is the square of the number of program points, i.e., the number of pairs considered in the analysis. $\mathbf{PPs^2} - \tilde{\mathcal{E}}_{\mathbf{P}}$ gives us the number of pairs that are guaranteed not to happen in parallel. $\mathbf{R}\varepsilon = 100(\tilde{\mathcal{E}}_{\mathbf{P}} - \mathbf{E_p})/\mathbf{PPs^2}$ is the approximated error percentage taking $\mathbf{E_p}$ as reference, i.e., $\mathbf{R}\varepsilon$ is an upper bound of the real error of the analysis. $\mathbf{T_G}$ is the time (in milliseconds) taken by the method-level analysis and in the graph construction. $\mathbf{T}_{\tilde{\mathcal{E}}_{\mathbf{P}}}$ is the time needed to infer all possible pairs of program points that may happen in parallel.

Although the MHP analysis has been successfully applied to both industrial case studies, it has not been possible to capture their runtime parallelism due to limitations in the simulator which could not treat all parts of these applications. Thus, there is no measure of error in these cases. We argue that the analyzer achieves high precision, with the approximated error less than 32.9% (bear in mind that $\mathbf{E_p}$ is a lower bound of the real parallelism) and up to 0% in other cases. As regards efficiency, both the method-level analysis and the graph construction are very efficient (just 0.36 sec. for the largest case study). The $\tilde{\mathcal{E}}_{\mathbf{P}}$ inference takes notably more time. But, as explained in Sec. 4.4, for most

applications only a subset of pairs is of interest and, besides, those pairs can be computed on demand.

6 Conclusions, Related and Future Work

We have proposed a novel and efficient approach to infer MHP information for concurrent objects. The main novelty is that MHP information is obtained by means of a local analysis whose results can be modularly composed by using a MHP *analysis graph* in order to obtain global MHP relations. Concurrent objects operate similarly to Actors [2] and Erlang processes [5]. Therefore, the main ideas of our approach could be adapted to these languages.

When compared to the MHP analysis for X10 proposed in [10, 1], we should first note that the async-finish model simplifies the inference of *escape* information, since the finish construct ensures that all methods called within its scope terminate before the execution continues to the next instruction. Moreover, it is important to note that our approach would achieve the same precision as their context-sensitive motivating example (Sec. 2.2 in [10]). This is because we do not merge calling contexts, but rather leave them explicit in the MHP graph. In addition, by splitting the analysis in two phases we achieve: (1) a higher degree of modularity and incrementality, since when a method is modified (or added, deleted, etc.), we only need to re-analyze that method locally, and replace its corresponding sub-graph in the global MHP graph accordingly; and (2) on demand MHP analysis, since we do not need to compute all MHP pairs in order to check if two given program points might run in parallel, but rather just check the relevant conditions for those two program points only.

An MHP analysis for Ada has been presented in [13], and extended later for Java in [14]. These works have been superseded later by [11, 6]. In [6], Java programs are abstracted to an *abstract thread model* which is then analyzed in two phases. MHP graphs are used as well despite being substantially different from ours. A main difference is that our first phase infers local information for each method, while that of [6] infers a thread-level MHP from which it is possible to tell which threads might *globally* run in parallel. In addition, unlike our method-level analysis, it does not consider any synchronization between the threads in the first phase, but rather in the second phase. In future work, we plan to investigate if our analysis can be adapted to this thread abstract model.

An important application of MHP analysis is for understanding if two program points that belong to different tasks *in the same object* might run in parallel (i.e., interleave). We refer to this information as object-level MHP. This information is valuable because, in any static analysis that aims at approximating the objects' states, when a suspended task resumes, the (abstract) state of the corresponding object should be refined to consider modifications that might have been done by other tasks that interleave with it. Our approach can be directly applied to infer object-level MHP pairs by incorporating points-to information [15, 12].

References

1. S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *PPOPP'07*, pages 183–193. ACM, 2007.
2. G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.
3. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *APLAS'11*, volume 7078, pages 238–254. Springer, 2011.
4. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *PEPM'12*, pages 151–154. ACM Press, January 2012.
5. J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
6. R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *LCPC'05*, volume 4339, pages 152–169. Springer, 2005.
7. F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *ESOP'07*, volume 4421, pages 316–330. Springer, 2007.
8. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO'10*, volume 6957, pages 142–164. Springer, 2012.
9. J. K. Lee, J. Palsberg, and R. Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.
10. J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *PPoPP'10*, pages 25–36, New York, NY, USA, 2010. ACM.
11. L. Li and C. Verbrugge. A practical mhp information analysis for concurrent java programs. In *LCPC'04*, pages 194–208. Springer, 2004.
12. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *ISSTA*, pages 1–11, 2002.
13. G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT FSE'98*, 23(6):24–34, 1998.
14. G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing *MHP* information for concurrent java programs. *ESEC / SIGSOFT FSE'99*, 24(6):338–354, 1999.
15. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*, pages 131–144. ACM, 2004.