A

May-Happen-in-Parallel Analysis for Actor-based Concurrency

Elvira Albert, Universidad Complutense de Madrid
Antonio Flores-Montoya, Technische Universität Darmstadt
Samir Genaim, Universidad Complutense de Madrid
Enrique Martin-Martin, Universidad Complutense de Madrid

This article presents a *may-happen-in-parallel* (MHP) analysis for languages with *actor-based concurrency*. In this concurrency model, actors are the concurrency *units* such that, when a method is invoked on an actor a_2 from a task executing on actor a_1, statements of the current task in a_1 may run in parallel with those of the (asynchronous) call on a_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. The information yielded by the MHP analysis is essential to infer more complex properties of actor-based concurrent programs, e.g., data race detection, deadlock freeness, termination and resource consumption analyses can greatly benefit from the MHP relations to increase their accuracy. We report on MayPar, a prototypical implementation of an MHP static analyzer for a distributed asynchronous language.

Categories and Subject Descriptors: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Actors, Analysis, Concurrency, May-Happen-in-Parallel

1. INTRODUCTION

We consider actor systems [Agha 1986; Haller and Odersky 2009], a model of concurrent programming that has been gaining popularity and that it is being used in many systems (such as ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala). Actor programs consist of computing entities called actors, each with its own local state and thread of control, that communicate by exchanging messages asynchronously. An actor configuration consists of the local state of the actors and a set of pending messages (or *tasks*). In response to receiving a message, an actor can update its local state, send messages, or create new actors. At each step in the computation of an actor system, an actor from the system is scheduled to process one of its pending messages.

Concurrent objects [Pierce 1994; Clarke et al. 2010] constitute an object-oriented implementation of actor systems. In particular, the concurrent objects model is based on considering objects as actors (i.e., the concurrency units), in such a way that each object conceptually has a dedicated processor. Communication is based on asynchronous method calls with objects as targets. An essential feature of actor-based systems 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

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.

DOI: http://dx.doi.org/10.1145/0000000.0000000

the source code and can be syntactically identified. Data-driven synchronization is possible by means of so-called *future* variables [de Boer et al. 2007] as follows. Consider an asynchronous method call m on object a, written as y=a.m(). Here, the variable y is a future which allows synchronizing with the result of executing task m. In particular, the instruction **await** y? allows checking whether m has finished, and lets the current task release the processor to allow another available task to take it. For simplicity, we assume that future variables are *local* to methods, i.e., we do not allow *inter-procedural* synchronization in which a task spawned in one task is awaited in a different task. We refer to [Albert et al. 2015] for the extension of the analysis to inter-procedural synchronization.

This article presents a novel *may-happen-in-parallel* (MHP) analysis for actor systems, and formalizes and implements it for a language based on concurrent objects. The goal of an MHP analysis is to identify pairs of statements that can execute in parallel. The MHP problem is known to be NP-complete for the *rendezvous* model of Ada [Taylor 1983], or even undecidable if procedure calls are allowed [Ramalingam 2000]. However several approaches have been proposed to make the analysis interprocedural [Duesterwald and Soffa 1991], expressing it as a data-flow analysis [Naumovich and Avrunin 1998] or extend it to object oriented languages like Java [Naumovich et al. 1999; Barik 2005] or X10 [Agarwal et al. 2007; Lee and Palsberg 2010]. In the context of concurrent objects, an asynchronous method invocation y_2=a_2.m() within a task t_1 executing in an object a_1 implies that the subsequent instructions of t_1 in a_1 may execute in parallel with the instructions of m within a_2. However, if the asynchronous call is synchronized with an instruction **await** y_2?, 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 y_3=a_3.p(); after 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 [Lee and Palsberg 2010] 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** (list !=**null**) {y=a.process(list.data); **await** y?; list = list.next;}, assuming list 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 list 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 executing 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 that 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 direct and through transitive calls– in the application. Then, the problem of inferring if two statements p_1 and p_2 can run in parallel amounts to checking certain *reachability* conditions between p_1 and p_2 in the MHP analysis graph.

1.1. Summary of Contributions

Our main contribution is the formalization and implementation of a novel MHP analysis for languages based on actor-based concurrency. Technically, the main contributions can be summarized as follows:

(1) We introduce a method-level analysis which infers the status of the tasks spawned within a method (ignoring transitive calls). The status of each task can be pending (i.e., waiting the be executed), active (i.e., executing) or finished (i.e., the return statement has been already executed). The analysis is formalized as a data-flow analysis where abstract states are symbolic values representing the status of tasks.
(2) We introduce an application-level analysis which composes the results obtained by the method-level analysis of all methods by building an MHP-graph. The graph contains nodes that represent program points and nodes that represent the status of methods. The key idea is that the status inferred by the method-level analysis determines the edges between these types of nodes of the graph. In particular, we connect the program points of a given method to the corresponding status of the tasks spawned in this method at this specific point. The resulting graph allows us to obtain direct and indirect MHP relations.
(3) We present an extension of our analysis to improve the overall precision in the presence of conditional statements. Instead of merging the information about spawned tasks in different branches after conditional statements, we keep separated the status of the tasks spawned in every possible path. Therefore we need to extend the representation of abstract states used in the method-level analysis and introduce a new kind of nodes in the MHP-graph obtained in the application-level analysis.
(4) We have implemented an MHP analyzer, named MayPar, for the ABS language. ABS [Johnsen et al. 2012] 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.

This article is an improved and extended version of the paper [Albert et al. 2012] that appeared in the proceedings of FORTE'12 and which is the basis for other analyses published later in [Flores-Montoya et al. 2013; Albert et al. 2013; Albert et al. 2014b]. The extensions consist in proving the soundness of the approach, in defining the partial MHP analysis and the accuracy improvements when dealing with conditional statements.

1.2. Organization of the Article

The article is organized as follows. Section 2 describes the syntax and the semantics of the concurrent objects language on which we develop our analysis. In order to be as generic as possible we focus on the concurrency and distribution features, while the syntax for expressions and types is left free.

Section 3 formally defines the property of may-happen-in-parallel that we want to then over-approximate by means of static analysis. This section also explains by means of examples the different types of MHP relations that we may have. The notion of *escaped tasks* is used to illustrate the problem of having MHP pairs with tasks that have been transitively invoked from a task that is known to have finished.

The core of the analysis is presented in Section 4 in four steps. We first formalize the method level analysis in Section 4.1 by defining a data-flow analysis an its underlying abstract states and operations. Then, in Section 4.2, we present the construction of the MHP-graph from which the MHP pairs are obtained. Section 4.3 defines the problem of inferring if two points can run in parallel by checking certain *reachability* conditions between the two points in the MHP analysis graph. Section 4.4 discusses the complexity of the analysis. Finally, Section 4.5 introduces the notion of points of interest and explains how the performance of the analysis can be improved in practice by considering a reduced amount of program points of interest depending on the application.

In Section 5 we present an extension of the analysis that mitigates the precision loss that sometimes happens when handling conditional statements in the original analysis. Sections 5.1 and 5.2 explains the modifications needed in the method-level and application-level analysis respectively—namely extend the representation of abstract states to store information from different paths of execution and introduce new nodes in the graph.

Our analysis has been the basis of recent work to develop other analyses that infer the complex properties of deadlock-freeness [Flores-Montoya et al. 2013], resource boundedness [Albert et al. 2013] and the peak resource consumption [Albert et al. 2014b]. Section 6 discusses the most direct application of our analysis to data race detection, as well as applications to proving deadlock-freeness, termination and resource analysis.

Section 7 describes our prototype implementation and the main results obtained by the experimental evaluation. Our implementation as well as the examples used in the article can be found online at `http://costa.ls.fi.upm.es/costabs/mhp`.

Section 8 overviews related work and Section 9 concludes and points out several directions for further research.

The proofs of all formal statements can be found in the appendix.

2. LANGUAGE

The actor-based paradigm [Agha 1986] 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. We consider a distributed message-passing program model in which each actor represents a processor which is equipped with a procedure stack and an unordered buffer of pending messages. Initially all actors are idle. When an idle actor's message buffer is non-empty, some message is removed, and a message-dependent task starts to execute. Each task besides accessing its own actor's global storage, can post messages to the buffers of any actor, including its own and synchronize with the reception of messages. When a task does complete, or when it is awaiting for a message that has

not arrived yet, its processor again becomes idle, chooses a next pending message to remove, and so on. Thus, we allow a concurrent behaviour within the tasks of each actor.

In our language, the concept of actor is materialized by means of an *object*. Tasks from different objects (i.e., different actors) execute in parallel. The distinction between messages and handling tasks is purely aesthetic, and we unify the two concepts by supposing that each message is a procedure-and-argument pair. Tasks can be synchronized with the completion of other tasks (from the same or a different actors) using futures. The number of actors does not have to be known a priori and objects can be dynamically created.

2.1. Syntax

A *program* consists of a set of classes, each of them can define a set of fields, and a set of methods. In addition, there is a method called main, not associated to any class, from which the execution starts. The grammar below describes the syntax of our programs. Here, T stands for types, m for method names, e for expressions, a for field accesses or local variables, and y for future variables. Future variables are local to methods and they cannot be passed in method parameters.

$$
\begin{array}{lll}
CL & ::= & class\ C\ \{\bar{T}\ \bar{f}; \bar{M}\} \\
M & ::= & T\ m(\bar{T}\ \bar{l})\ \{\bar{T}\ \bar{l'}; s\} \\
s & ::= & \epsilon \mid instr; s \\
instr & ::= & \mid a{=}e \mid \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \textbf{do}\ s \mid \textbf{return}\ a \mid \\
 & & \textbf{await}\ y? \mid \textbf{release} \mid a{=}\textbf{new}\ C(\bar{e}) \mid y{=}a.m(\bar{e}) \mid a{=}y.\textbf{get}
\end{array}
$$

The notation $\bar{T}\ \bar{f}$ is used as a shorthand for the sequence $T_1\ f_1; \ldots; T_n\ f_n$, where T_i is a type and f_i is a field name. Similarly, $\bar{T}\ \bar{l}$ is used for declaring the formal parameters of a method, and $\bar{T}\ \bar{l'}$ is used for declaring its local variables. We use the special identifier **this** to denote the current object, for example, **this**$.m(\bar{e})$ is a call to a method m in the object on which the instruction is executing. Note that the above syntax forbids assigning values to future variables y, except in the instruction $y{=}a.m(\bar{e})$. We assume that every method has at least one **return** instruction, and thus each method must have at least one instruction. Note that in the assignment $a{=}e$, we assume that a is not a future variable. This choice simplify the presentation as it allows us to avoid incorporation of may-aliasing information.

We assume that future variables are local, that is, they cannot be declared as fields or passed around as parameters or returned by methods. For the sake of generality, the syntax of expressions and types is left free. In fact, in the rest of this article, the only knowledge on types (and expression) we need is to distinguish future variables from others, which can be done syntactically depending on the instruction under consideration. We let $P_{\mathcal{M}}$ and $P_{\mathcal{F}}$ stand for the set of method and future variable names, respectively, in the program P.

Each object represents a processor and has a heap with the values assigned to its fields. The concurrency model is as follows. Each object has a lock that is shared by all tasks that belong to the object. Data synchronization is by means of future variables as follows. An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}a.m(\bar{e})$ such that **await** $y?$ is executed only when the future variable y is available (and hence the task executing m on object a is finished). In the meantime, the object's lock can be released and some *pending* task on that object can take it. The instruction $a_2{=}y.$**get** blocks the object (no other task of the same object can run) until y is available, that is, the execution of $m(\bar{e})$ on a is *finished*, and stores the return value in the variable a_2. Note that the difference from **await** $y?$ is in that it *blocks* the object. The instruction **release** unconditionally yields the object's lock so that other pending task can take it.

(NEWOBJECT)
$$\frac{\mathsf{fresh}(bid')\ ,\ l' = l[a \to bid'], f' = init\_atts(C, \bar{e})}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle a\text{=}\textbf{new}\ C(\bar{e}); s\rangle) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l', s),\ obj(bid', f', \bot)}$$

(SELECT)
$$\frac{s \neq \epsilon(v)}{obj(bid, f, \bot),\ tsk(tid, bid, \_, \_, s) \rightsquigarrow obj(bid, f, tid),\ tsk(tid, bid, \_, \_, s)}$$

(ASYNC)
$$\frac{l(a) = bid_1,\ \mathsf{fresh}(tid_1),\ l' = l[y \to tid_1],\ l_1 = buildLocals(\bar{e}, m_1)}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle y\text{=}a.m_1(\bar{e}); s\rangle) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l', s),\ tsk(tid_1, bid_1, m_1, l_1, body(m_1))}$$

(AWAIT1)
$$\frac{l(y) = tid_1,\ s_1 = \epsilon(v)}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{await}\ y?; s\rangle),\ tsk(tid_1, \_, \_, \_, s_1) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l, s),\ tsk(tid_1, \_, \_, \_, s_1)}$$

(AWAIT2)
$$\frac{l(y) = tid_1, s_1 \neq \epsilon(v)}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{await}\ y?; s\rangle),\ tsk(tid_1, \_, \_, s_1) \rightsquigarrow \\ obj(bid, f, \bot),\ tsk(tid, bid, m, l, \langle \textbf{await}\ y?; s\rangle),\ tsk(tid_1, \_, \_, s_1)}$$

(GET)
$$\frac{l(y) = tid_1, s_1 = \epsilon(v), l' = l[a \to v]}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle a\text{=}y.\textbf{get}; s\rangle),\ tsk(tid_1, \_, \_, \_, s_1) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l', s),\ tsk(tid_1, \_, \_, \_, s_1)}$$

(RELEASE)
$$\frac{}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{release}; s\rangle) \rightsquigarrow \\ obj(bid, f, \bot),\ tsk(tid, bid, m, l, s)}$$

(RETURN)
$$\frac{v = l(a)}{obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{return}\ a; s\rangle) \rightsquigarrow \\ obj(bid, f, \bot),\ tsk(tid, bid, m, l, \epsilon(v))}$$

Fig. 1. Summarized Semantics

We use *release point* to refer to lines in a program containing **await** $y?$, **release** or **return** a instructions, as at those points the object's lock may be released.

Note that our concurrency model is *cooperative* as processor release points are explicit in the code, in contrast to a *preemptive* model in which a higher priority task can interrupt the execution of a lower priority task at any point. Without loss of generality, we assume that all methods in a program have different names.

2.2. Semantics

A *program state* $St = \texttt{Obj} \cup \texttt{Tsk}$ is composed by a set of objects `Obj` and a set of tasks `Tsk`. Each task from `Tsk` is associated to an object from `Obj`. Each *object* is a term $obj(bid, f, lk)$ where bid is the object identifier, f is a mapping from the object fields to their values, lk is the identifier of the *active task* that holds the object's lock or $\bot$ if the object's lock is free. A *task* is a term $tsk(tid, bid, m, l, s)$ where tid is a unique task identifier, bid is the object identifier to which the task belongs, m is the method name executing in the task, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and its return value v is available. Only one task can be *active* (running) in each object and has its *lock*, so objects can be seen as *monitors* [Buhr et al. 1995]. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. Created objects and tasks never disappear from the state.

The execution of a program starts from its main method. Namely, from initial state $S_0 = \{obj(0, [], 0), tsk(0, 0, \mathsf{main}, l, body(\mathsf{main}))\}$ where $obj(0, [], 0)$ is an auxiliary object

that has no fields, and $tsk(0, 0, \mathsf{main}, l, body(\mathsf{main}))$ is a task associated to this object and is executing method main. Here, l maps (initial) parameters to their initial values and local reference and future variables to **null** (standard initialization), and $body(\mathsf{main})$ refers to the sequence of instructions in the method main. Program execution is *nondeterministic*, i.e., given a state there may be different execution steps that can be taken, depending on the object selected. Furthermore, when an object's lock is released, it chooses non-deterministically any pending task in the queue to continue. The execution proceeds from S_0 by non-deterministically applying one of the semantic rules depicted in Figure 1. We omit the treatment of the sequential instructions as it is standard and, moreover, is not required for our analysis. Next we explain the different rules of Figure 1:

— NEWOBJECT: An active task tid in object bid creates an object bid' of type C, its fields are initialized ($init\_atts$) and bid' is introduced to the state with a free lock.
— SELECT: This rule selects non-deterministically one of the tasks that is in queue and is not finished, and its object lock is current free, and it obtains its object's lock.
— ASYNC: A method call creates a new task (the initial state is created by *buildLocals*) in the corresponding object and with a fresh task identifier tid_1 which is associated to the corresponding future variable y in l'.
— AWAIT1 and AWAIT2: They deal with synchronization on future variables. If the future variable we are awaiting for points to a finished task then the **await** can be completed (AWAIT1); otherwise, the task yields the lock so that any other task of the same object can take it (AWAIT2).
— GET: It waits for the future variable but without yielding the lock. Then, it retrieves the value associated with the future variable.
— RELEASE: It unconditionally yields the lock.
— RETURN: When **return** is executed, the return value is stored in v so that it can be obtained by the future variable that points to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding $\epsilon(v)$) but it does not disappear from the state as its return value may be needed later.

3. DEFINITION OF MHP

We first formally define the concrete property *may-happen-in-parallel* that we want to overapproximate. For the sake of simplicity, in the rest of this article we assume a given program P.

In what follows, we assume that instructions are labeled 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, which is typically that of its first instruction, and $p_{\dot{m}}$ to refer to an exit program point which is reached when executing a **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, and we let $pp(\epsilon(v)) = p_{\dot{m}}$. Using $pp(s)$ we can define the *active program points* in a state.

Definition 3.1 (*active program point*). A program point p is active in a state S within task tid, if and only if there is a task $tsk(tid, \_, \_, \_, s) \in S$ such that $pp(s) = p$.

We sometimes say that p is active in S without referring to the corresponding task identifier. Intuitively, this means that there is a task in S whose next instruction to be executed is the one at program point p.

A	B	C	D	E

`int m() {`
`1 ...`
`2 y=a.p();`
`3 z=a.q();`
`4 ...`
`5 await z?;`
`6 ...`
`7 await y?;`
`8 ...`
`9 return` e_1`;`
`10}`
	`int m() {`
	`11 ...`
	`12 y=this.r();`
	`13 z=a1.p();`
	`14 z=a2.p();`
	`15 z=a3.q();`
	`16 w=z.get;`
	`17 ...`
	`18 await y?;`
	`19 return` e_2`;`
	`20}`
		`int m() {`
		`21 ...`
		`22 while b do`
		`23 y=a.q();`
		`24 await y?;`
		`25 z=a.p();`
		`26 ...`
		`27 ...`
		`28 ...`
		`29 return` e_3`;`
		`30}`
			`int m() {`
			`31 ...`
			`32 if b then`
			`33 y=a.p();`
			`else`
			`34 y=a.q();`
			`35 ...`
			`36 await y?;`
			`37 ...`
			`38 return` e_4`;`
			`39}`
				`int p() {`
				`40 y=a.r();`
				`41 ...`
				`42 return` e_5
				`43}`
				`int q() {`
				`44 y=a.r();`
				`45 await y?;`
				`46 ...`
				`47 return` e_6`;`
				`48}`

Fig. 2. Simple examples for different MHP behaviours.

Definition 3.2 (*concrete MHP*). The concrete MHP set of P is defined as $\mathcal{E}_P = \cup\{\mathcal{E}_S \mid S_0 \rightsquigarrow^* S\}$ where $\mathcal{E}_S = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, \_, \_, \_, s_1), tsk(tid_2, \_, \_, \_, s_2) \in S, tid_1 \neq tid_2\}$.

Intuitively, $\mathcal{E}_P$ is the set of pairs of program points that can be active simultaneously. Observe in the above definition that, as execution is non-deterministic, 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 depicted in Figure 2. The code fragments in the rest of the paper will use the following convention to avoid confusion about variables: local variables will use the letters a, b, c ... and future variables the letters y, z and w. There are 4 versions of a method m which call 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). The global MHP behavior of executing each m (separately) is as follows:

(A) both p and q are called from m, then r is called from p and q. The **await** instruction at program point 5 (L5 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** instruction at L45. The **await** instruction at L7 waits until p has finished before continuing. This means that at L8, p is no 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) both q and p are called from m, but p is called twice. Any program point of p, for example L41, might execute in parallel with q even if they do not call each other, that is, they have an *indirect* MHP relation. Furthermore, L41 might execute in parallel with any program point of m after the method call at L13. 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. Method r is called at L12, however, as r belongs to the same object as m, it will not be able to start executing until m reaches the release point at L18. We say that r is *pending* from L13 up to L17.

(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** instructions, q will not execute in parallel with itself. At L27, we might have any number of p instances executing

but none of q. Note that if any method escaped from q, it might execute in parallel with L27.

(D) The last example illustrates an **if** statement. At L35, either p or q are executing but they cannot run in parallel even if m is a common ancestor. Furthermore, after the **await** instruction (L36) neither q or p are executing. This information will be extracted from the fact that both calls use the same future variable.

4. MHP ANALYSIS

In this section we develop an MHP analysis which statically overapproximates the concrete MHP set $\mathcal{E}_P$. The analysis is done in two main steps, first it infers method-level MHP information, for each method m separately, which basically tells which methods, from those invoked directly in m, might be executing at each program point. Then, in order to obtain application-level MHP, it composes this information by building an MHP graph from which the required global MHP information can be obtained.

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, namely, it is at the entry program point; (2) *finished*, which means that it has finished executing already, namely, 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 exit). As we explain later, the distinction between these statuses is essential for precision.

The underlying abstract states used in the analysis are sets of symbolic values, that describe the status of all tasks invoked so far. There are symbolic values to describe a single task in one of the states described above, and there are other values to describe several (one or more) tasks with the same characteristics (e.g., several instances of a method f in a pending state). The analysis itself can be seen as an abstract symbolic execution that collects the abstract states at each program point. Intuitively, when a method is invoked, we add it to the set, and its status will be pending or active depending if it is a call on the same object or on a different object; when an **await** $y?$ or a=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** $y?$, **release** or **return**), we change the status of all pending methods to active. Next we apply this intuition to the programs of Figure 2.

Example 4.1. Consider programs A and B in Figure 2. In A, the call to p at L2 creates an *active* task that becomes *finished* at L8. Similarly, the call to q at L3 creates an *active* task that becomes *finished* at L6. In B, the call to r at L12 creates a *pending* task that becomes *active* at L18 and *finished* at L19. The calls to p at L13 and L14 create tasks that are *active* up to the end of the method. These tasks will never become *finished* as their associated future variable is reused in L15 to synchronize with q.

In the rest of this section we formalize the method-level analysis, following the intuition described above, as follows: (1) we start by defining the abstract values and some corresponding operations, such as computing an upper bound; (2) we define a transfer function that describes the (abstract) effect of executing each instruction; and (3) we finally describe how to build a system of data-flow equations whose solutions provide us with the desired information, namely, which methods might be executing, and in which state they are, at each program point. Note that this last step corresponds to the symbolic execution that we have mentioned above.

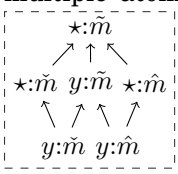
Definition 4.2 (*MHP atoms*). A *single* MHP atom is a symbolic expression of the form $y{:}\tilde{m}$, $y{:}\hat{m}$ or $y{:}\check{m}$, where $m \in P_{\mathcal{M}}$ is a method name and y is either a future variable from $P_{\mathcal{F}}$ or the special symbol $\star$. Similarly, a *multiple* MHP atom is a symbolic expression of the form $(\star{:}\tilde{m})^+$, $(\star{:}\hat{m})^+$ or $(\star{:}\check{m})^+$. We refer to both kinds as MHP atoms. The set of all MHP atoms is defined as

$$\mathcal{A} = \{y{:}x \mid m \in P_{\mathcal{M}}, x \in \{\tilde{m}, \hat{m}, \check{m}\}, y \in P_{\mathcal{F}}\} \cup \{\star{:}x, (\star{:}x)^+ \mid m \in P_{\mathcal{M}}, x \in \{\tilde{m}, \hat{m}, \check{m}\}\}\,.$$

Let us explain the intended meaning of the different MHP atoms:

(1) $y{:}\tilde{m}$ describes an *active* task that is an instance of method m;
(2) $y{:}\hat{m}$ describes a *finished* task that is an instance of method m;
(3) $y{:}\check{m}$ describes a *pending* task that is an instance of method m; and
(4) $(\star{:}x)^+$ is used to represent multiple occurrences of $\star{:}x$, i.e., it is used to describe a situation where several tasks of the same kind are running in parallel.

Note that in all cases above, a task is associated to a future variable y. When it is not possible to determine to which future variables a task is associated, e.g., if they are reused or assigned in a loop, we use $\star$ to represent any future variable. Note that multiple atoms always use $\star$, while simple ones might use future variable y or $\star$.



The elements of $\mathcal{A}$ are partially ordered with respect to the partial order relation $\preceq$ that we define next. The diagram depicted on the left defines when $y_1{:}x_1 \preceq y_2{:}x_2$ holds. This is then lifted to other kinds of atoms as follows: $(\star{:}x_1)^+ \preceq (\star{:}x_2)^+$ if $\star{:}x_1 \preceq \star{:}x_2$, and $y{:}x_1 \preceq (\star{:}x_2)^+$ if $y{:}x_1 \preceq \star{:}x_2$. Given a, $a' \in \mathcal{A}$ the meaning of $a \preceq a'$ is that concrete scenarios described by the MHP atom 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.

Definition 4.3 (*abstract MHP states*). An abstract MHP state M is a set of MHP atoms from $\mathcal{A}$. The set of all sets over $\mathcal{A}$ is denoted by $\mathcal{B}$.

Intuitively, each $y{:}x \in M$ represents *one* task that *might* be available and is associated to future variable y (or any future variable if $y = \star$). The status of the task is active, pending or finished, respectively, if $x = \tilde{m}$, $x = \check{m}$ or $x = \hat{m}$. Similarly, each $(\star{:}x)^+ \in M$ represent several tasks of the same kind (i.e., $\star{:}x$) that *might* be available. Note that when several tasks are associated to the same future variable, 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).

The elements of $\mathcal{B}$ are partially ordered with respect to a partial order $\sqsubseteq$ that we define next. 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, we define $M_1 \sqsubseteq M_2$ if all elements of M_1 are covered by elements from M_2 such that each single MHP atom $x{:}y \in M_2$ covers at most one MHP atom form M_1.

Example 4.4. Consider programs A, B and D in Figure 2. The sets $\{y{:}\tilde{p}, z{:}\tilde{q}\}$, $\{y{:}\tilde{p}, z{:}\hat{q}\}$, $\{y{:}\hat{p}, z{:}\hat{q}\}$, $\{y{:}\check{r}, z{:}\tilde{p}\}$, $\{y{:}\check{r}, (\star{:}\tilde{p})^+, z{:}\hat{q}\}$ and $\{y{:}\tilde{p}, y{:}\tilde{q}\}$ respectively describe the abstract states at L4, L6, L8, L14, L17 and L35. An important observation is that, in the abstract state of L17, 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 different execution paths and are incomparable, i.e., $a \not\preceq a'$ and $a' \not\preceq a$, as in L35.

Next we define an effective procedure for computing an upper bound $M_3 \in \mathcal{B}$ for a given abstract states $M_1, M_2 \in \mathcal{B}$, that is, an abstract state M_3 such that $M_1 \sqsubseteq M_3$ and $M_2 \sqsubseteq M_3$. This is mainly required to join abstract states at program points in which several execution paths meet, e.g., loop entries and program points after conditional

Algorithm 1: Upper Bound Procedure for Abstract MHP States

$M_1 \sqcup M_2$
Input: Abstract MHP states M_1 and M_2
Output: An upper bound M_3 for M_1 and M_2
begin

1 Remove $a \in M_i$ from M_i if there is $(\star{:}x)^+ \in M_i$ and $a \preceq \star{:}x$ // for $i = 1, 2$
2 $M_i' := \{(\star{:}x)^+ \mid (\star{:}x)^+ \in M_i\}$ // for $i = 1, 2$
3 $M_i := M_i \setminus M_i'$ // for $i = 1, 2$
4 $M_3 := M_1' \cup M_2'$
5 $M_i := M_i \setminus \{y{:}x \mid y{:}x \in M_i, (\star{:}x')^+ \in M_3, y{:}x \preceq \star{:}x'\}$ // for $i = 1, 2$
6 $M_4 := M_1 \cap M_2$
7 $M_i := M_i \setminus M_4$ // for $i = 1, 2$
8 $M_3 := M_3 \cup M_4$
9 **foreach** $y{:}x \in M_1$ **do**
10 **if** $\exists y'{:}x' \in M_2$ *such that* $y'{:}x' \preceq y{:}x$ *or* $y{:}x \preceq y'{:}x'$ **then**
11 $M_1 := M_1 \setminus \{y{:}x\}$
12 $M_2 := M_2 \setminus \{y'{:}x'\}$
13 $M_3 := M_3 \cup \{a\}$ where $a = y{:}x$ if $y'{:}x' \preceq y{:}x$ and $a = y'{:}x'$ if $y{:}x \preceq y'{:}x'$
14 $M_3 := M_3 \cup M_1 \cup M_2$
15 **return** M_3

statements. Note that typically one is interested in the least upper bound, however, as we show in the following example, it might not exit in some cases.

Example 4.5. Let $M_1 = \{\star{:}\hat{m}, \star{:}\check{m}\}$ and $M_2 = \{\star{:}\tilde{m}\}$. Both $M_3 = \{\star{:}\hat{m}, \star{:}\tilde{m}\}$ and $M_3' = \{\star{:}\tilde{m}, \star{:}\check{m}\}$ 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.

Clearly one can take the union $M_1 \cup M_2$, as an upper bound, however, it is not precise in many cases. Our procedure for computing an upper bound $M_3 \in \mathcal{B}$ for $M_1, M_2 \in \mathcal{B}$ is depicted in Algorithm 1, and it works as follows:

(1) In Line 1, we remove redundant elements from M_1 and M_2, if there is any.
(2) In lines 2–4, any *multiple* MHP atom in M_1 (respectively M_2) is added to M_3 and removed from M_1 (respectively M_2). The intuition is that these atoms cannot be covered by any other atom since they represent unknown number of occurrences of some single MHP atoms;
(3) In Line 5, the atoms of M_1 and M_2 that are covered by a multiple atom of M_3 are removed from their respective sets;
(4) In lines 6–8, the atoms $M_1 \cap M_2$ are added to M_3, and removed from M_1 and M_2;
(5) In lines 9–13, each pair of atoms $y{:}x \in M_1$ and $y'{:}x' \in M_2$ such that $y'{:}x'$ covers $y{:}x$ (or vice versa) are removed from their respective sets, and the bigger one is added to M_3. Note that there are several possible ways to compute the covering as we have seen before, the code at lines 9–13 is just a possible one; and
(6) In Line 14, $M_1 \cup M_2$ is added to M_3.

Note that due to the way Step 5 above is implemented, and since there is no unique way to compute the covering, it might be the case that $M_1 \sqcup M_2 \neq M_2 \sqcup M_1$. Note that $\sqcup$ is lifted to compute the upper bound of several MHF states in standard way. The following Lemma states the soundness of the procedure described in Algorithm 1.

$$
\begin{array}{lrl}
(1) & \texttt{kill}(y{=}a.m(\bar{e})) = & \{y{:}x \mid q \in P_{\mathcal{M}}, x \in \{\check{q}, \tilde{q}, \hat{q}\}\} \\
(2) & \texttt{kill}(y{=}\textbf{this}.m(\bar{e})) = & \text{as (1)} \\
(3) & \texttt{kill}(\textbf{release}) = & \{y{:}\check{q} \mid y \in P_{\mathcal{F}}, q \in P_{\mathcal{M}}\} \cup \{\star{:}\check{q}, (\star{:}\check{q})^{+} \mid q \in P_{\mathcal{M}}\} \\
(4) & \texttt{kill}(a{=}y.\textbf{get}) = & \{y{:}x \mid q \in P_{\mathcal{M}}, x \in \{\check{q}, \tilde{q}\}\} \\
(5) & \texttt{kill}(\textbf{await}\ y?) = & \text{as (4)} \\
(6) & \texttt{kill}(\textbf{return}\ a) = & \text{as (3)} \\
(7) & \texttt{kill}(a = e) = & \emptyset \\
(8) & \texttt{kill}(e) = & \emptyset \\
\hline
(1) & \texttt{gen}(y{=}a.m(\bar{e}), M) = & \{y{:}\tilde{m}\} \cup \\
 & & \{\star{:}x \mid q \in P_{\mathcal{M}}, x \in \{\check{q}, \tilde{q}, \hat{q}\}, y{:}x \in M \wedge \star{:}x \notin M\} \cup \\
 & & \{(\star{:}x)^{+} \mid q \in P_{\mathcal{M}}, x \in \{\check{q}, \tilde{q}, \hat{q}\}, y{:}x \in M \wedge \star{:}x \in M\} \\
(2) & \texttt{gen}(y{=}\textbf{this}.m(\bar{e}), M) = & \text{as (1) but the first disjunct is } \{y{:}\check{m}\} \\
(3) & \texttt{gen}(\textbf{release}, M) = & \{y{:}\tilde{q} \mid q \in P_{\mathcal{M}}, y \in P_{\mathcal{F}}, y{:}\check{q} \in M\} \cup \\
 & & \{\star{:}\tilde{q} \mid q \in P_{\mathcal{M}}, \star{:}\check{q} \in M \wedge \star{:}\tilde{q} \notin M\} \cup \\
 & & \{(\star{:}\tilde{q})^{+} \mid q \in P_{\mathcal{M}}, (\star{:}\check{q})^{+} \in M \vee (\star{:}\check{q} \in M \wedge \star{:}\tilde{q} \in M)\} \\
(4) & \texttt{gen}(a{=}y.\textbf{get}, M) = & \{y{:}\hat{q} \mid q \in P_{\mathcal{M}}, y{:}\tilde{q} \in M\} \\
(5) & \texttt{gen}(\textbf{await}\ y?, M) = & \text{as (4)} \\
(6) & \texttt{gen}(\textbf{return}\ a, M) = & \text{as (3)} \\
(7) & \texttt{gen}(a = e, M) = & \emptyset \\
(8) & \texttt{gen}(e, M) = & \emptyset
\end{array}
$$

Fig. 3. $\texttt{kill} : s \mapsto \mathcal{B}$ and $\texttt{gen} : s \times \mathcal{B} \mapsto \mathcal{B}$ functions

LEMMA 4.6. *Let $M_1, M_2 \in \mathcal{B}$ and $M_3 = M_1 \sqcup M_2$, then $M_1 \sqsubseteq M_3$ and $M_2 \sqsubseteq M_3$.*

The correctness of the above lemma is straightforward, it is easy to see that Algorithm 1 constructs M_3 such that (1) every MHP atom from M_1 (resp. M_2) is covered by one MHP atom from M_3; and (2) every single MHP atom from M_3 covers at most one MHP atom from M_1 (resp. M_2).

Next we define the effect of executing each instruction on a given abstract state. This is done by the following transfer function

$$
\begin{array}{l}
\tau : s \times \mathcal{B} \mapsto \mathcal{B} \\
\tau(I, M) = (M \setminus \texttt{kill}(I)) \cup \texttt{gen}(I, M)
\end{array}
$$

where $\texttt{kill}(I)$ and $\texttt{gen}(I, M)$ are as defined in Figure 3. Intuitively, $\texttt{kill}(I)$ are the elements to be removed from M and $\texttt{gen}(I, M)$ are the elements to be added to M. Let us explain the different cases of `kill` and `gen`:

(1) when asynchronously calling a method m using $y{=}a.m(\bar{e})$, function `kill` removes all elements that are associated with future variable y in M, since y is rewritten. Then, function `gen` adds the atom $y{:}\tilde{m}$ to indicate that m can be executing at any program point (because a might be an object different from the one executing the call, and thus might start to execute immediately), and that it is associated to a future variable y. In addition, for each atom that has been removed by `kill`, i.e., those that used future variable y, `gen` adds a corresponding one that uses $\star$ (if the corresponding $(\star{:}x)^{+}$ is not yet in M).

(2) when asynchronously calling a method m using $y{=}\textbf{this}.m(\bar{e})$, m will execute on the same object that is executing the call, which means that m will not start to execute until the current task reaches a release point. Thus, this case is identical to one above except that `gen` adds $y{:}\check{m}$ instead of $y{:}\tilde{m}$ to indicate that m is pending.

(3) when executing **release**, the current task releases the lock allowing other tasks to take it. In particular, this allows pending tasks to become active. Thus, `kill`

removes any atom that corresponds to a pending task, and function gen adds corresponding active atoms.

(4) when executing $a{=}y.$**get**, the current task blocks and waits to a method that is associated to future variable y to finish, and in the meanwhile it does not release the lock. This means that when the current task resumes we are sure that the method associated to y has finished, and that all other tasks running in the same object preserve their (abstract) status since the lock is not released. Thus, function kill simply removes pending and active atoms that are associated to y, and function gen adds corresponding finished ones.

(5) the **await** $y?$ instruction is special since it encapsulates two instructions: when future variable y is not ready it behaves like a **release**, and when it is ready it simply continues to the next instruction. The transfer function handles the second case, which is actually equivalent to **get**, the first one will be simulated by the data-flow equation that we generate later in this section. Another alternative, which does not require special treatment for **await**, is to assume that the program is instrumented such that every **await** instruction is preceded by a **release**.

(6) when executing **return**, since the current task terminates, any other task can take the lock of the current object. Thus, its abstract behavior is the same as **release**.

(7) executing an assignment $a{=}e$, where a is not a future variable, does not alter the abstract state.

(8) evaluating an expression e, in **if** and **while**, does not alter the abstract state.

Note that the transfer function τ function might generate states with redundant elements, e.g., both $y{:}x$ and $(y{:}x)^+$ are included in the result. In such case one may assume that they are removed in a post-processing step (avoiding this in the definitions of kill and gen would unnecessarily make them complex).

Example 4.7. Consider program B of Figure 2. The abstract state at L12 is $\emptyset$ since we have not invoked any method yet. Executing L12 adds $y{:}\check{r}$ since the call is to a method in the same object; executing L13 adds $z{:}\tilde{p}$; executing L14 changes $z{:}\tilde{p}$ to $\star{:}\tilde{p}$ (since the future variable z is reused) and adds $z{:}\tilde{p}$ again; executing L15 changes $z{:}\tilde{p}$ to $\star{:}\tilde{p}$ and adds $z{:}\tilde{q}$; executing L16 changes $z{:}\tilde{q}$ to $z{:}\hat{q}$ since it is guaranteed that q has finished; The abstract state at L18 is obtained by applying the transfers function of **release** on the abstract state of L17 (assuming L17 does not alter the abstract state), this changes $y{:}\check{r}$ to $y{:}\tilde{r}$, since the current task might suspend and thus any pending task might become active. Note again that this is not explicit in the definition of the transfer function of **await** $y?$, it will become clear when we generate the corresponding data-flow equations below. Finally, we apply the case of **await** $y?$ on the abstract state of L18, which in turn changes $y{:}\tilde{r}$ to $y{:}\hat{r}$.

Next we describe how to build a system of data-flow equations [Nielson et al. 2005, Ch. 2] whose solutions describe the desired method-level MHP information. First we fix some notation. We let I_p denote the instruction at program point p, such that if p corresponds to an **if** or **while** then I_p refers to evaluating the corresponding condition. We let $\texttt{pre}(p)$ be the set of program points that immediately precede program point p, that is, when they are executed they immediately reach p, in particular $\texttt{pre}(p_{\dot{m}})$ is the set all program points $p \in m$ such that $I_p =$ **return** a. Note that if p is a loop entry, then the program point of the last instruction of the loop body is included in $\texttt{pre}(p)$. We let $\texttt{init}(P)$ and $\texttt{final}(P)$ be the sets of all entry and exit program points of P, respectively.

Definition 4.8. The set of method-level MHP data-flow equations, denoted by $\mathcal{L}_{\mathcal{P}}$, includes the following two equations for each program point $p \in P_{\mathcal{P}}$:

$$\mathcal{L}_\circ(p) = \begin{cases} \emptyset & \text{if } p \in \texttt{init}(P) \\ \tau(\textbf{release}, \bigsqcup_{p' \in \texttt{pre}(p)} \mathcal{L}_\bullet(p')) & I_p \equiv \textbf{await} \\ \bigsqcup_{p' \in \texttt{pre}(p)} \mathcal{L}_\bullet(p') & \text{otherwise} \end{cases}$$

$$\mathcal{L}_\bullet(p) = \tau(I_p, \mathcal{L}_\circ(p)) \qquad p \notin \texttt{final}(P)$$

Intuitively, each program point p contributes two equations: (1) $\mathcal{L}_\circ(p)$ captures the MHP information *before* executing the instruction I_p, which simply merges the MHP states of all preceding programs points except for the **await** instruction since it encapsulates a **release** instruction as well; and (2) $\mathcal{L}_\bullet(p)$ captures the MHP state *after* executing the instruction I_p, which simply applies the corresponding case of the transfer function on I_p and $\mathcal{L}_\circ(p)$. Note that for the exit program points $\texttt{final}(P)$ we do not generate $\mathcal{L}_\bullet(p)$ since there is no instructions at such program points.

Example 4.9. The method-level MHP data-flow equations for the programs of Figure 2 are depicted in Figure 4. Note that L1, L11, L21, L31, L40 and L44 are method entry points, and L10, L20, L30, L39, L43 and L48 are method exit points.

There are several standard algorithms for solving data-flow equations (see for example Nielson et al. [2005, Sect. 2.4] and Aho et al. [1986, Sect. 9.3]). These algorithms are based, in principle, on computing the least fixpoint of an abstract semantic operator f, that simply evaluates the right hand side of the data-flow equation using the current approximation, using Kleene iterations. In practice, this is done by starting from an initial state in which, using our notation, $\mathcal{L}_\circ(p)$ and $\mathcal{L}_\bullet(p)$, for all $p \in P_{\mathcal{P}}$, are mapped to $\emptyset$, and then in each iteration the current values of $\mathcal{L}_\circ(p)$ and $\mathcal{L}_\bullet(p)$ are used to compute new values by substituting them on the right hand side of the equations (after each iteration the old values are replaced by the new ones). This is repeated until a (least) fixpoint is reached. Correctness of these algorithms rely on that the underlying domain has a least upper-bound operation, and that the transfer function is monotone. Without these properties these algorithms are not guaranteed to generate ascending chains of (abstract) elements and thus might not terminate, even for finite domains [Gange et al. 2013].

While it is easy to see that our transfer function τ is monotone, our domain does not have a least upper bound as we have seen in Example 4.5, and, moreover, our upper bound operation as defined in Figure 1 is not even monotone, e.g., for M_1, M_2, M_3 and M_3' of Example 4.5 we have $M_2 \sqsubseteq M_3'$ but $M_1 \sqcup M_2 = M_3 \not\sqsubseteq M_3' = M_1 \sqcup M_3'$. To overcome this problem, the semantic operator f is typically replaced by $g(x) = \lambda x.x \sqcup f(x)$ so that *old and new values are merged* in each iteration instead of staying with the new one [Cousot and Cousot 1979]. In practice, this modification amounts to applying the following steps in each iteration:

(1) for each equation $a = exp$ evaluate exp using the current approximations. Let the result be a';
(2) set each a to $a \sqcup a'$;
(3) if no a has changed its value, we are done; otherwise go to 1.

This iterative process, applied to our equations, always terminate since: (i) $\mathcal{B}$ does not have infinite ascending chains; and (ii) the values assigned to each a are non-decreasing.

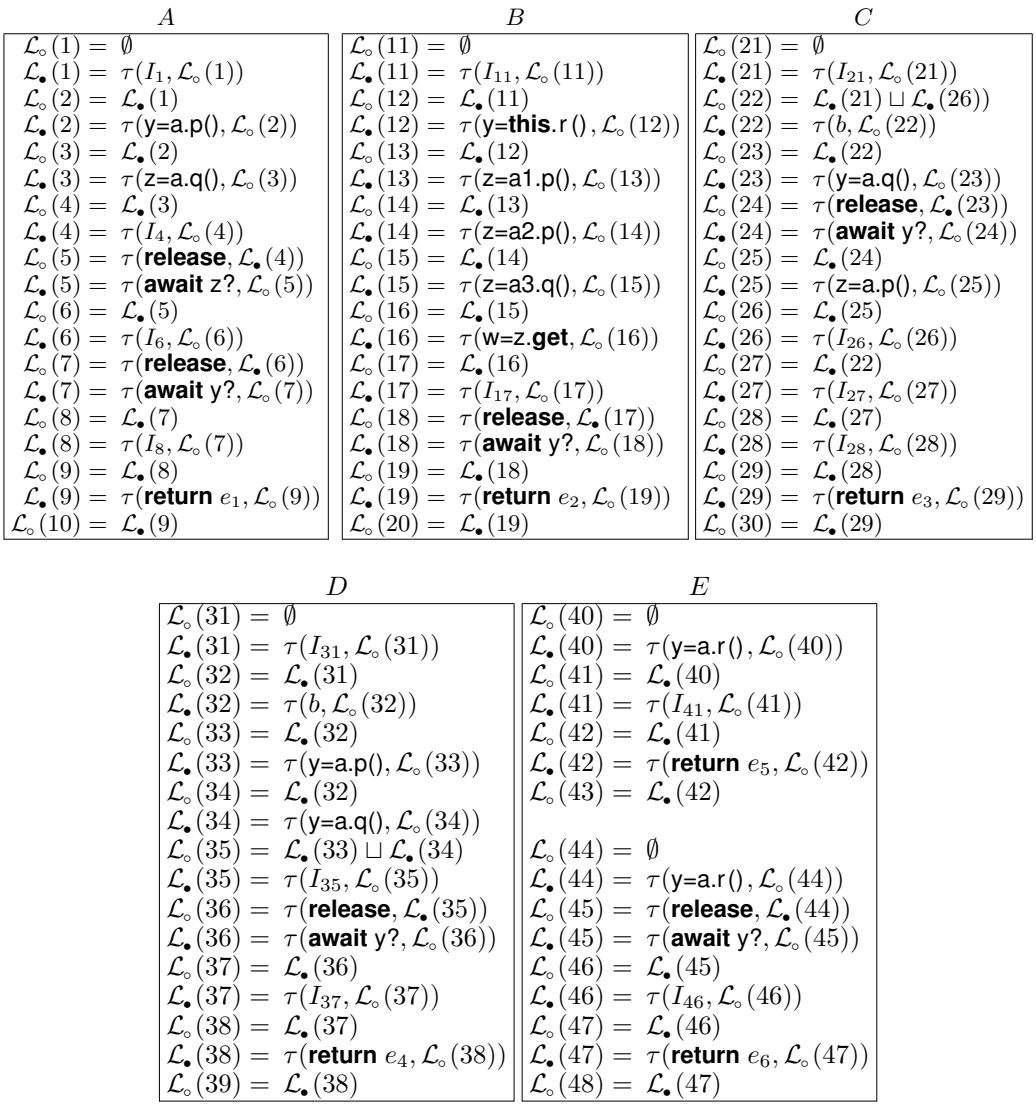


Fig. 4. Method-level MHP data-flow equations for the examples of Figure 2

Example 4.10. Solving the equations of Example 4.9, that are depicted in Figure 4 results in the solution depicted in Figure 5. Note that in some sets redundant elements were removed, e.g., $y{:}\hat{q}$ was removed from $\mathcal{L}_\bullet(24)$ because of $(\star{:}\hat{q})^+$, and, $z{:}\tilde{p}$ was removed from $\mathcal{L}_\bullet(25)$ because of $(\star{:}\tilde{p})^+$.

Let us finish this section with a discussion on aliasing of local future variables and how it affects the correctness of the analysis. Recall that in our language we have assumed that in the assignment $a{=}e$, variable a is not a future variable. Under this assumption aliasing of future variables is not possible, and thus it was completely ignored. A natural question to ask now is how the method-level MHP analysis would be affected if we allow instructions that introduce aliasing between future variables,

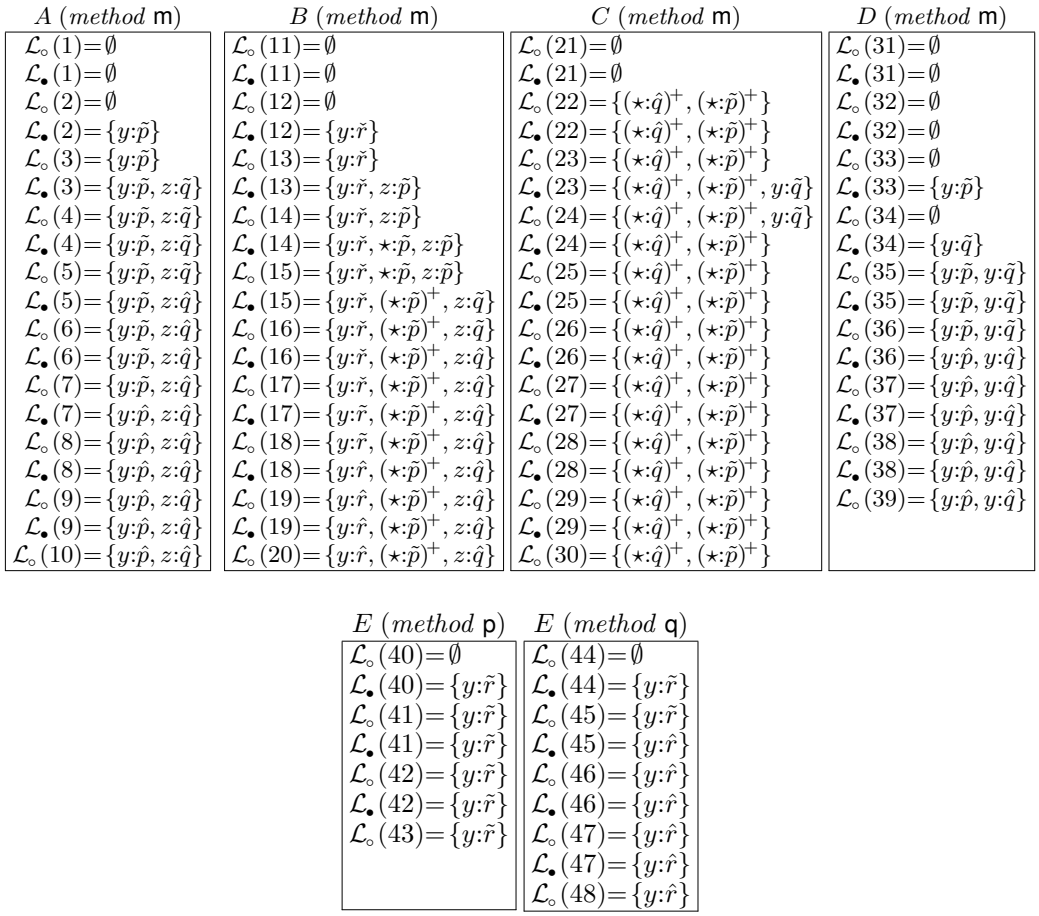


Fig. 5. A solution of the method-level MHP data-flow equations Figure 4

e.g., $z = y$. Next we show that the soundness of the analysis is preserved as long as we adequately define the transfer function for future variable assignments $z = y$. The transfer function of $z = y$ must simply overwrite z.

We first explain the meaning of an abstract state M, in particular, what does it mean that M correctly overapproximates the concrete scenarios (a formal justification is available in the Appendix). Let M be the *abstract* method-level MHP information for program point p. The first question that should be clarified is: what is the *concrete* method-level information that M approximates? For this, we consider an arbitrary reachable state St that includes a task executing at program point p, i.e., $t = tsk(tid, bid, m, l, s) \in St$ and $p = pp(s)$, and define the *concrete* method-level information of t as the set of tasks $T \subseteq St$ that were created due to method calls performed by t (before reaching St). Now for M to correctly overapproximate the concrete method-level information at p, it should correctly describe the tasks in T, namely:

> For any task $t' = tsk(tid', bid', m', l', s') \in T$ there is an MHP atom in M that covers t', i.e., correctly describes its status, and if this atom is associated

with a future variable y, then we actually have $l(y) = tid'$. Moreover, any single MHP atom can describe only one $t' \in T$.

Note that the above must hold for any reachable state St, and any $t \in St$ that is executing at program point p. Now we claim that in order to support aliasing in our analysis, we only need to define a case for $z = y$ in the transfer function that replaces all occurrences of future variable z by $\star$:

$$\begin{aligned} \texttt{kill}(z\texttt{=}y) &= \{z{:}x \mid q \in P_{\mathcal{M}}, x \in \{\check{q}, \tilde{q}, \hat{q}\}\} \\ \texttt{gen}(z\texttt{=}y, M) &= \{\star{:}x \mid q \in P_{\mathcal{M}}, x \in \{\check{q}, \tilde{q}, \hat{q}\}, z{:}x \in M\} \end{aligned}$$

This might be of course imprecise, since we do not track the aliasing, however, as we explain next, it is sound.

Assume an abstract state M that correctly overapproximates the local MHP at program point p, and assume that the program point p' immediately follows p. If we execute $z = y$, then at the concrete level we do not create any new task, and at the abstract level we just replace occurrences of z by $\star$, so any task that was covered by $z{:}x \in M$ will be covered by $\star{:}x$ in the new abstract state. This shows that our treating of $z = y$ is sound. Similar reasoning can be applied to all other instructions, which is actually done in the proofs (see the Appendix) since there we do not assume anything about aliasing. Next we summarize this reasoning, showing that $M' = \tau(I_p, M)$ correctly overapproximates the concrete method-level MHP information at p', independently from any aliasing between variables:

— In the case a method calls $y = a.m(\bar{e})$ (resp. $y =$ **this**$.m(\bar{e})$), at the concrete level we create a new task that is associated to future variable y, and at the abstract level a new atom $y{:}\tilde{m}$ (resp. $y{:}\check{m}$) is added, and it clearly covers the new task. All other tasks are covered as in M, except that a task that was covered by $y{:}x \in M$ is now covered by $\star{:}x \in M'$.
— In the case of **release**, at the concrete level it changes *some* pending tasks to active, and at the abstract level we change *all* pending tasks to active. Thus, if a concrete task was covered by $y{:}\check{m} \in M$, it will be covered by $y{:}\tilde{m} \in M'$ since the active status include pending one as well.
— In the case of $a = y$.**get**, at the concrete level the status of the task bounded to y, call it t, changes to finished, so at p this task must be either finished or active (if it is pending the execution never moves to p', because it is executing in the same object, and thus it does not matter what happens at the abstract level). If t was covered by $\star{:}\tilde{m} \in M$ or $\star{:}\hat{m} \in M$, then this same atom will be in M' and can be used to cover the task t at p'. If t was covered by $y{:}\tilde{m} \in M$ or $y{:}\hat{m} \in M$, then $y{:}\hat{m} \in M'$ can be used to cover the t in at p'.
— The case of **await** $y?$ is as of $a = y$.**get**, the case of **return** a is as **release** and all other cases are straightforward since they do not alter the abstract state.

We conclude that, as long as z is overwritten for every future variable assignment of the form $z = y$, the soundness of the analysis is preserved. Although our analysis is sound in the presence of aliasing, the treatment of aliasing as described above can be imprecise in some scenarios, however, we can use must-aliasing information to solve some related imprecision problems as we show in the next example.

Example 4.11. Let us assume an abstract state $M = \{y{:}\tilde{m}, z{:}\tilde{q}\}$, which describes a scenario in which we might have two tasks executing and pointed to by future variables y and z. Executing $z = y$ within M results in $M' = \tau(z = y, M) = \{y{:}\tilde{m}, \star{:}\tilde{q}\}$. Note that in M' future variable z is not associated with any task, and thus if we execute $a = z$.**get** within M' we get $M'' = \tau(a = z.\textbf{get}, M') = M'$. This is clearly imprecise because after executing $a = z$.**get**, the task described by $y{:}\tilde{m}$ becomes finished since y and z alias,

and thus a precise analysis would change $y{:}\tilde{m}$ to $y{:}\check{m}$. To overcome this imprecision we could modify the transfer function, for the cases of $a = y.\textbf{get}$ and **await** $y?$, to change any MHP atom that uses a future variable that must-alias with y to finished.

4.2. The Notion of MHP Graph

Next we 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 Definition 3.2). An 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. Edges from method nodes to program points represent points of which at most one might be executing. In contrast, edges from a program point node to method nodes represent that some of those methods (possibly all of them) might be running at that specific program point. The information computed by the method-level MHP analysis is required to construct the MHP graph, in particular for constructing the out-edges of program point nodes. Edges that correspond to multiple MHP atoms will be labeled by ∞.

We start by formally constructing the MHP graph, and then explain the construction in detail. We assume that the set of method-level MHP equations $\mathcal{L}_{_P}$ has been generated and solved, in particular, for a program point p we assume that the value of $\mathcal{L}_\circ(p)$ is available.

Definition 4.12 (*MHP graph*). The MHP graph of program P is a directed weighed 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:

$$\begin{aligned}
V &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{_\mathcal{M}}\} \cup P_{_\mathcal{P}} \cup \{p_y \mid p \in P_{_\mathcal{P}}, y{:}x \in \mathcal{L}_\circ(p), y \neq \star\}\\
E_1 &= \{\tilde{m} \to p \mid m \in P_{_\mathcal{M}}, p \in m\} \cup \{\hat{m} \to p_{\hat{m}}, \check{m} \to p_{\check{m}} \mid m \in P_{_\mathcal{M}}\}\\
E_2 &= \{p \to x \mid p \in P_{_\mathcal{P}}, \star{:}x \in \mathcal{L}_\circ(p)\} \cup \{p \overset{\infty}{\to} x \mid p \in P_{_\mathcal{P}}, (\star{:}x)^+ \in \mathcal{L}_\circ(p)\}\\
E_3 &= \{p \to p_y, p_y \to x \mid p \in P_{_\mathcal{P}}, y{:}x \in \mathcal{L}_\circ(p)\}
\end{aligned}$$

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 variable $y \neq \star$ that appears in $\mathcal{L}_\circ(p)$ contributes a node p_y. These nodes will be used to state that if there are several MHP atoms in $\mathcal{L}_\circ(p)$ 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) $\tilde{m} \to p$ for each $p \in m$, which means that if m is active it can be any program point – but only at one; (b) $\check{m} \to p_{\check{m}}$, which means that when m is pending, it is at the entry program point; and (c) $\hat{m} \to p_{\hat{m}}$, 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 at each program point. For every program point $p \in P_{_\mathcal{P}}$, if $\star{:}x \in \mathcal{L}_\circ(p)$ (resp. $(\star{:}x)^+ \in \mathcal{L}_\circ(p)$) then $p \to x$ (resp. $p \overset{\infty}{\to} x$) is added to E_2. Such edge means, if $x = \tilde{m}$ for example, that an instance (resp. several instances) of m might be running in parallel when reaching p. Note

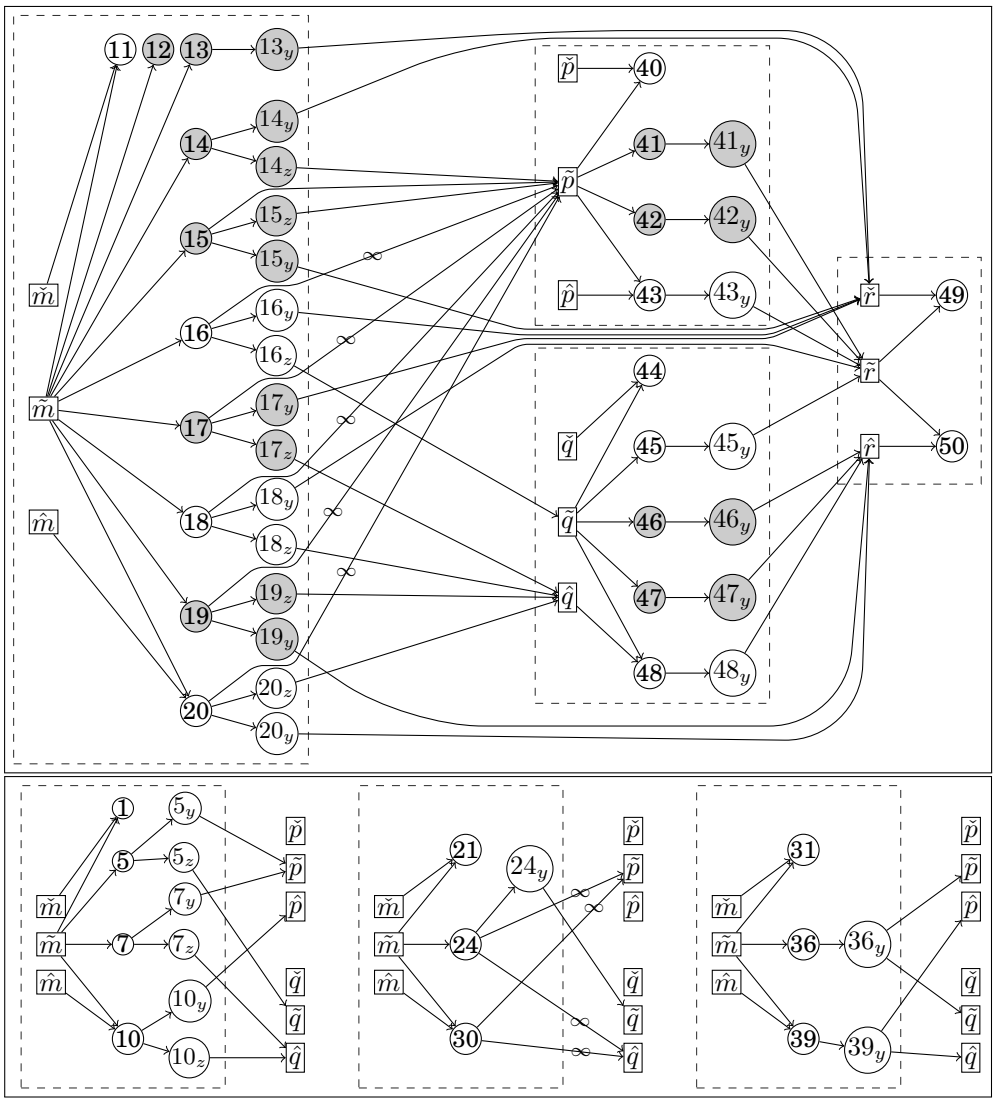


Fig. 6. The MHP graphs of the examples of Figure 2: (**top**) the graph of Example B, together with the graphs of methods q, p and r; (**bottom**) the graphs of examples A, C and D, from left to right, where the omitted part for methods q, p and r is the same as the top one. The graph of each method is surrounded by a dashed box.

that edges labeled with ∞ can be seen, in principle, as multiple edges (at least two) between the corresponding nodes;

(3) Edges in E_3 enrich the information for each program point given in E_2. An edge $p_y \rightarrow x$ is added to E_3 if $y{:}x \in \mathcal{L}_\circ(p)$. For each future variable y that appears in $\mathcal{L}_\circ(p)$ an edge $p \rightarrow p_y$ is also added to E_3. This allows us to accurately handle cases in which several MHP atoms in $\mathcal{L}_\circ(p)$ are associated to the same future variable.

Note that MHP graphs might have cycles due to recursion.

Example 4.13. Consider again the programs of Figure 2. Using the method-level MHP information of Example 4.10, we obtain the MHP graphs depicted in Figure 6. The graph at the top corresponds to Example B, which also includes the graphs of methods q, p and r. We assume that the entry and exit program points of method r are 49 and 50 respectively. The gray nodes play no particular rule here, they will be considered in Section 4.5. The graphs at the bottom correspond to examples A, C and D, from left to right, respectively. In these graphs the parts that correspond to methods q, p and r were omitted for readability, they are the same as the one on top. Besides, for readability, the graphs of A, C and D do not include all program points, but rather only those that correspond to entry, **get**, **await**, and exit program 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, that is, 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 is labeled with ∞ (since such edge can be seen as multiple edges).

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task that was invoked within the task executing p_1 and from which it is possible to *transitively* reach p_2, or vice-versa. This is the case, for example, of L16 and L45 in Figure 6 (top). The second case corresponds to *indirect MHP* scenarios in which a task is running at p_3, and there are two other tasks that were invoked within the task executing p_3 and from which it is possible to reach p_1 and p_2. This is the case, for example, of L42 and L46 that are both reachable from L16 in Figure 6 (top), through paths that start with different edges. Observe that the first edge can only be shared if it is labeled with ∞ because it represents that there might be more than one instance of the same type of task running. This allows us to infer that L41 may run in parallel with itself because the edge from L16 to $\tilde{p}$ has weight ∞, and, besides, that L41 can run in parallel with L42. Note that L44-L48 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 $p_1 \to x \rightsquigarrow p_2$ to indicate that such path starts with an edge to x. Note that the edge from p_1 to x might be labeled with ∞ (we ignore the label when it is not relevant).

Definition 4.14. The MHP information induced by the MHP graph $\mathcal{G}_P$ is defined as $\tilde{\mathcal{E}}_P = \mathit{directMHP} \cup \mathit{indirectMHP}$ where

$$\begin{aligned} \mathit{directMHP} = \quad & \{(p_1,p_2),(p_2,p_1) \mid p_1,p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P)\} \\ \mathit{indirectMHP} = \cup\ & \{(p_1,p_2) \mid p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \to x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \to x_2 \rightsquigarrow p_2 \in \mathcal{G}_P, x_1 \neq x_2\} \\ & \{(p_1,p_2) \mid p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \overset{\infty}{\to} x_1 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \overset{\infty}{\to} x_1 \rightsquigarrow p_2 \in \mathcal{G}_P\} \end{aligned}$$

Example 4.15. The table depicted in Figure 7 represents some of the pairs in $\tilde{\mathcal{E}}_P$ obtained from the graph of Figure 6 (top). Empty cells mean that the corresponding points cannot run in parallel. Cells marked by • indicate that the pair is in $\mathit{directMHP}$. Cells marked with ∘ indicate that the pair is in $\mathit{indirectMHP}$. Note that the table captures the MHP relations informally discussed in Section 3.

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

	11	16	18	20	40	43	44	45	48	49	50
11											
16					●	●	●	●	●	●	●
18					●	●			●	●	●
20					●	●			●	●	●
40		●	●	●	○	○	○	○	○	○	○
43		●	●	●	○	○	○	○	○	●	●
44		●			○	○				○	○
45		●			○	○				●	●
48		●	●	●	○	○				○	●
49		●	●	●	○	●	○	●	○	○	○
50		●	●	●	○	●	○	●	●	○	○

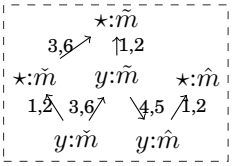
Fig. 7. $\tilde{\mathcal{E}}_P$ for some program points of interest, obtained from the MHP graph of Figure 6.

THEOREM 4.16 (SOUNDNESS). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

Let us finish this section with some remarks regards precision. *Escape information* is essential for the MHP analysis, as it collects which of the tasks created in one method are still alive when the method terminates. That information is stored in the exit program point of the method, and it is used when computing the global MHP pairs—notice that an exit program point generates a node in the MHP graph with edges to the escaping methods.

4.4. Complexity of the Analysis

As regards the complexity of the analysis, we distinguish its three phases: the method-level analysis, the generation of the MHP graph $\mathcal{G}_P$, and the computation of the MHP pairs $\tilde{\mathcal{E}}_P$ from $\mathcal{G}_P$.



Generating and solving the method-level MHP constraints can be performed independently for each method m. If we consider the abstract states as multisets where $(y{:}m)^+$ represents two or more single atoms, we can see that there are two types of operations in the transfer function. The operations that add new atoms (1) and (2) (See Fig. 3) and the operations that transform some atoms into others (3-6). Given an atom that is added to the abstract state, it can be transformed a limited number of times according to the diagram on the left. In particular, an atom can be transformed at most 3 times.

We take the result of [Hecht and Ullman 1973; Kam and Ullman 1976] into account. Given a rapid data flow problem, if we adopt an iterative algorithm in which nodes in the control graph are visited in reverse postorder, the information is propagated in $d+1$ iterations and the fixpoint is guaranteed after $d+2$ iterations where d is defined as the maximum degree of nested loops in the control flow graph.

Our domain is not rapid, but given that the atom addition in (1) and (2) is unconditional (independent of the previous state at that point), in every iteration a new copy of the same atoms is created. These copies need $d+1$ iterations to propagate to all the control flow graph. However, during the propagation, the atoms can be transformed. We can assume as a worst case, that they are transformed in the last iteration of the propagation and the transformed atom has to be propagated again. Because they can be transformed up to 3 times, we need at most $(1+3)*(d+1)$ iterations to completely propagate the atoms created in the first iteration and their transformed versions.

Because we consider at most 2 instances of each single atom, when the atoms created in the second iteration are completely propagated, the state will be saturated (all the possible instances of $(y{:}m)^+$ must be already present) and we will have reached the fixpoint. In the next iteration we will be able to confirm the fixpoint. In conclusion, we need at most $4 * (d + 1) + 2$ iterations.

In addition, it is possible to represent abstract states M such that the cost of all operations is linear with respect to their sizes which are at most in $O(nm_m \cdot \mathit{fut}_m)$, 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 generating and solving $\mathcal{L}_P$ for a method m is in $O(d \cdot pp_m \cdot nm_m \cdot \mathit{fut}_m)$ where pp_m is the number of program points in the method. Note that the procedure for computing an upper bound of Figure 1 can be implemented in linear time as well. This is because in order to check if a given atom $a \in M_1$ is covered by an atom $a' \in M_2$, we do not need to traverse all elements of M_2, but rather only those that refer to the same method and the same future variable (or $\star$). Therefore, with an appropriate data structure this can be done in linear time. The used data structure is an array of nm_m elements in which each element contains the information of a method m' that can be called. Such information is three naturals for the number of atoms $\star{:}x$ such that $x \in \{\tilde{m'}, \check{m'}, \hat{m'}\}$ and three arrays of size fut_m for representing the active, pending and finished atoms for each future variable.

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 $O(pp'_m \cdot nm_m \cdot \mathit{fut}_m)$ 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, this will be discussed in Section 4.5.

Once the graph has been created, computing $\tilde{\mathcal{E}}_P$ is basically a graph reachability problem. Therefore, an algorithm for inferring $\tilde{\mathcal{E}}_P$ is in $O(n^3)$, where n is the number of nodes of the graph. We can use the Floyd-Warshall algorithm for transitive closure to compute all-pairs reachability, and using this pre-computed information we can check whether (p_1, p_2) is a MHP pair or not in $O(n^2)$. However, a major advantage of our 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*.

4.5. Partial analyses and points of interest

As mentioned before, in many cases the interest is not in a complete set of MHP pairs, but rather is restricted to some program points of interest such us those correspond to **release**, **await**, and **get** instructions. In this section we show how, for such cases, the overall performance can be improved by safely discarding some program points when building the corresponding MHP graph.

We first define the notion of program points of interest, partial MHP analysis and a sufficient condition for a program point to be safely ignored. This condition does not take into account the instruction at that program point, but rather defined in terms of the result of the method-level analysis. Then, we reexamine this condition from the perspective of the instruction being executed at that program point, and extract those instruction that are essential, that is, for which the corresponding program points should be included in the MHP graph.

Let $iP_{\mathcal{P}} \subseteq P_{\mathcal{P}}$ be the set of program points of interest, the partial MHP analysis of P with respect to $iP_{\mathcal{P}}$ aims at inferring MHP pairs that are relevant to program points from $iP_{\mathcal{P}}$.

Definition 4.17 (*partial MHP information*). The partial MHP information of P with respect to $iP_{\mathcal{P}}$ is defined as $p\tilde{\mathcal{E}}_P = \tilde{\mathcal{E}}_P \cap (iP_{\mathcal{P}} \times iP_{\mathcal{P}})$.

The partial MHP information consists of those pairs that involve program points from $iP_{\mathcal{P}}$. Our interest is to compute $p\tilde{\mathcal{E}}_P$ directly, and not by computing $\tilde{\mathcal{E}}_P$ and then restricting it to $p\tilde{\mathcal{E}}_P$ as in the above definition. Recall that in Example 4.15, we actually were interested in a subset of the program points, however, in order to compute them we have used an MHP graph that includes all program points.

The partial MHP analysis is similar to the MHP analysis developed so far. The first phase, namely, the method-level analysis, is the same and must consider all program points. The difference is in the second phase, which constructs the MHP graph taking into account only a subset of the program points (maybe larger than $iP_{\mathcal{P}}$).

Intuitively, we could ignore every program point $p \in P_{\mathcal{P}}$ that does not belong to $iP_{\mathcal{P}}$ and does not add new information to the analysis. If a program point serves as a link between two points in $iP_{\mathcal{P}}$ and is the only one we will not be able to ignore it. The following is a sufficient condition for a program point to be ignored—the proof can be found in the appendix.

LEMMA 4.18. *Let $p \in m$ be a program point such that $p \notin iP_{\mathcal{P}}$. If there is a program point $p' \in m$, different from p, such that $\mathcal{L}_\circ(p) \sqsubseteq \mathcal{L}_\circ(p')$, then p can be safely ignored with respect to $iP_{\mathcal{P}}$.*

Intuitively, since all program points of m are connected in the same way to $\tilde{m}$, then if we remove p we must guarantee that there is another point p' from which we will be able to generate those paths removed due to removing p.

Example 4.19. Consider the MHP graph of Figure 6 (top), and in particular the nodes that correspond to L11-L16. One can easily verify that any program point node that is reachable from the nodes that correspond to L11-L15, is also reachable from the node that corresponds to L16. Indeed, from Example 4.10 we know that $\mathcal{L}_\circ(16)$ is larger than $\mathcal{L}_\circ(11)$, $\mathcal{L}_\circ(12)$, $\mathcal{L}_\circ(13)$, $\mathcal{L}_\circ(14)$, and $\mathcal{L}_\circ(15)$. Thus, if we are not interested in MHP pairs that involve L11-L15 then we can simply eliminate their corresponding nodes in the MHP graph.

Let us see how in practice we identify a program point that satisfy Lemma 4.18, depending on the instruction at that program point, and without examining the solution of $\mathcal{L}_{\mathcal{P}}$. Let s be a sequence of instructions in a method m and $pp(s) = p$. If $s = instr; s'$ and $pp(s') = p'$, by the definition of the transfer function τ, we have $\mathcal{L}_\circ(p) \sqsubseteq \mathcal{L}_\circ(p')$ for all instructions that are not **await** $y?$ or $a{=}y$.**get**. That is, the method-level information always grows except for **await** $y?$ and $a{=}y$.**get** instructions. If we apply Lemma 4.18, it is easy to see that it is safe to ignore all points except the exit $p_{\dot{m}}$, and those that correspond to **await** $y?$ or $a{=}y$.**get** instructions. Obviously, program points that are in $iP_{\mathcal{P}}$ cannot be ignored.

Example 4.20. Consider again program A of Figure 2, together with methods p, q, and r, and assume that our interest is in the set of program point that appear in the table of Figure 7. In such case, instead of considering the complete MHP graph of Figure 6 (top), we can consider one that does not include any of the gray nodes. They can be safely ignored since they are not in $iP_{\mathcal{P}}$, and do not correspond to exit program points or to **await** $y?$ or $a{=}y$.**get** instructions.

5. A MORE PRECISE HANDLING OF CONDITIONAL STATEMENTS

The MHP analysis of Section 4 handles conditional statements by merging the abstract states of the different branches at the end of the condition, using the upper bound procedure of Algorithm 1. This choice leads to an efficient analysis, however, it might lead to a loss of precision as well.

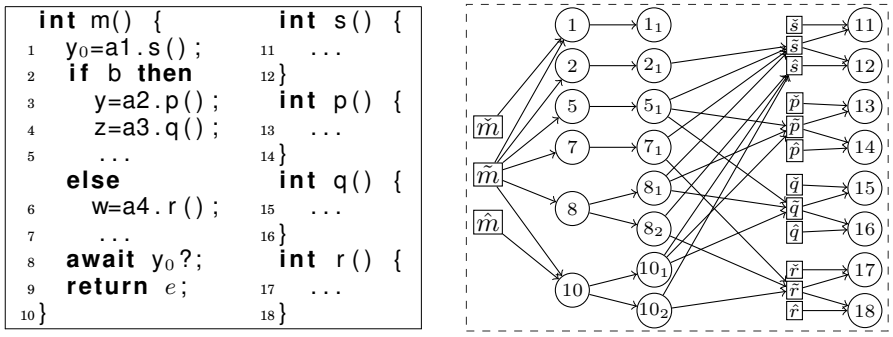


Fig. 8. An example with a conditional statement, and its corresponding (conditional) MHP graph $\mathcal{G}_P^H$.

Example 5.1. Consider program D of Figure 2, and assume that the instructions at L33 and L34 use different future variables, say y and z, moreover, assume that at L36 we do not have an **await** instruction. In such case, the abstract states at L35 and L36, inferred by the method-level analysis, would both be $\{y{:}\tilde{p}, z{:}\tilde{q}\}$. Then, in the corresponding MHP graph we have two different paths that start at the node of L36, one leads to $\tilde{p}$ (through the node of future variable y) and the other leads to $\tilde{q}$ (through the node of future variable z). Thus, according to Definition 4.14, we would say that any two program points of p and q might run in parallel, which is clearly not the case.

Similar loss of precision occurs also in the program depicted in Figure 8. Here method m calls s and, depending on condition b, it either calls methods p and q or method r. Clearly, p and q may run in parallel, but they will not run in parallel with r because they are in exclusive branches. However, the method-level analysis of Section 4.1 will merge the abstract states of both branches at L8 and obtain $\{y_0{:}\tilde{s}, y{:}\tilde{p}, z{:}\tilde{q}, w{:}\tilde{r}\}$. Now since the MHP atoms in this state do not share the same future variable, the global analysis of Section 4.3 will infer that any pair of program points of p and r, as well as q and r, might run in parallel.

The above example demonstrates that as far as the method calls in the branches of a conditional statements synchronize with the same future variable, the analysis of Section 4 can precisely handle them. However, if they synchronize with different future variables then it losses precision. In fact, the later case is very common since the ABS language [Johnsen et al. 2012], for which our analysis is implemented, allows calling methods without association to future variables.

In this section we present a modification of the MHP analysis of Section 4 to overcome this kind of imprecision. In Section 5.1 we modify the method-level analysis to use a new form of abstract states that allows keeping MHP information of different execution paths separately, and, in Section 5.2 we modify the construction of the MHP graph to take advantage of this new form of abstract states.

5.1. Method-level Analysis

In this section we modify the method-level analysis of Section 4.1 to use *sets of sets of MHP atoms* for representing abstract states, instead of *sets of MHP atoms*. This allows keeping the information of different execution paths separately.

Example 5.2. Consider the program depicted in Figure 8. Using *sets of sets of MHP atoms* for representing abstract states, for L8 we will obtain the abstract state $H = \{\{y_0{:}\tilde{s}, y{:}\tilde{p}, z{:}\tilde{q}\}, \{y_0{:}\tilde{s}, w{:}\tilde{r}\}\}$. The inner sets of H represent two exclusive scenarios that correspond to the different branches.

The modification of the method-level analysis to the new setting requires: defining a new set of abstract states; defining an upper bound operator for such states; adapting the transfer function to such setting; and generate the method-level MHP data-flow equations using these new elements.

As we have mentioned above, a (conditional) abstract MHP state in this section will be represented by a *set of sets of MHP atoms*. The set of all such abstract states is denoted by $\mathcal{H}$, and is partially ordered as follows:

$$H_1 \sqsubseteq H_2 \Leftrightarrow_{def} \forall M_1 \in H_1.\ \exists M_2 \in H_2.\ M_1 \sqsubseteq M_2$$

Note that M_1 and M_2 are abstract states as in Section 4. The upper bound of $H_1, H_2 \in \mathcal{H}$, denoted $H_1 \sqcup H_2$, is defined as the set union $H_1 \cup H_2$. The transfer function over the new form of abstract states is defined by lifting of the one of Figure 3 as follows

$$\tau_H(b, H) = \{\tau(b, M_i) \mid M_i \in H\}$$

The generation and solving of the method-level MHP equations is very similar to the one in Section 4.1, however, using the new operation for upper bound and the new transfer function $\tau_H(b, H)$.

Definition 5.3. The set of method-level MHP data-flow equations, denoted by $\mathcal{L}_P^H$, includes the following two equations for each program point $p \in P_{\mathcal{P}}$. Recall that $\mathcal{L}_\circ^H(p)$ captures the MHP information *before* executing the instruction I_p, whereas $\mathcal{L}_\bullet^H(p)$ captures that information *after* executing the instruction.

$$\mathcal{L}_\circ^H(p) = \begin{cases} \{\emptyset\} & \text{if } p \in \texttt{init}(P) \\ \tau_H(\textbf{release}, \bigsqcup_{p' \in \texttt{pre}(p)} \mathcal{L}_\bullet^H(p')) & I_p \equiv \textbf{await} \\ \bigsqcup_{p' \in \texttt{pre}(p)} \mathcal{L}_\bullet^H(p') & \text{otherwise} \end{cases}$$

$$\mathcal{L}_\bullet^H(p) = \tau_H(I_p, \mathcal{L}_\circ^H(p)) \qquad p \notin \texttt{final}(P)$$

Example 5.4. Generating and solving the method-level MHP equations for the program of Figure 8 generates the results depicted in Figure 9. Note that the abstract states of both branches (L3–L5 and L6–L7) evolve separately starting from the common state at L2, and they are joined at L8.

Example 5.5. Figure 10 shows a program with more complex uses of conditional statements. The method m1 contains two nested conditional expressions whose evaluation is controlled by the Boolean expressions b_1 and b_2. There are 3 possible paths: 1) b_1=**true**,b_2=**true**; 2) b_1=**true**,b_2=**false** and 3) b_1=**false**. Each path will generate a different pattern of calls, and the extended analysis will keep separately the information from the 3 different scenarios. Generating and solving the method-level MHP data-

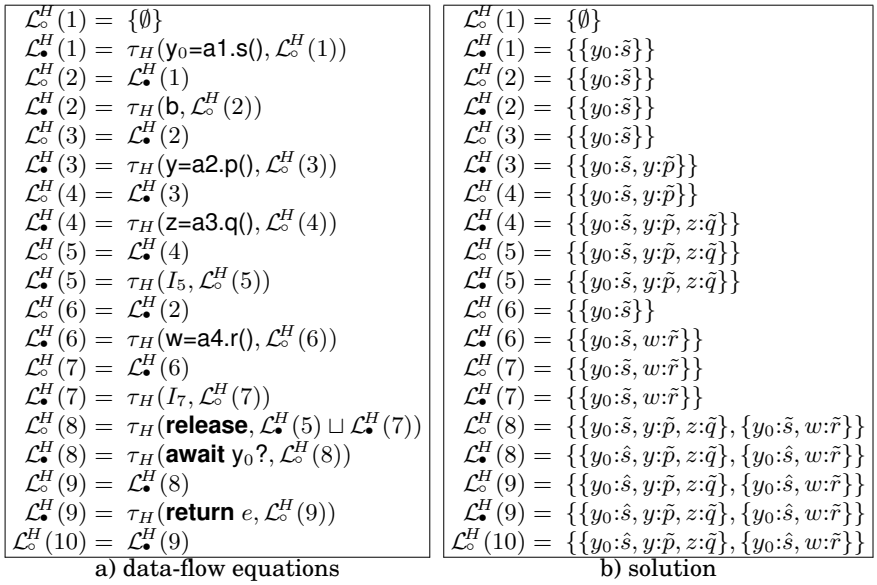


Fig. 9. Method-level MHP data-flow equations for the program in Figure 8

```
int m1() {
1   if b1 then
2     y0 = this.s();
3     if b2 then
4       y = a2.p();
5       ...
      else
6       z = a3.q();
7       ...
8     a = 3;
    else
9     w = a4.r();
10    ...
11  return e;
12}

int m2() {
13  ...
14  while b1 do
15    ...
16    if b2 then
17      this.p();
18      ...
      else
19      x.q();
20      ...
21    ...
22  return e;
23}

int s() {
24  ...
25}
int p() {
26  ...
27}
int q() {
28  ...
29}
int r() {
30  ...
31}
```

Fig. 10. An example with nested conditional statements and conditionals inside loops.

flow equations for m1 results in:

$\mathcal{L}^H_\circ(1)=\{\emptyset\}$
$\mathcal{L}^H_\bullet(1)=\{\emptyset\}$
$\mathcal{L}^H_\circ(2)=\{\emptyset\}$
$\mathcal{L}^H_\bullet(2)=\{\{y_0{:}\check{s}\}\}$
$\mathcal{L}^H_\circ(3)=\{\{y_0{:}\check{s}\}\}$
$\mathcal{L}^H_\bullet(3)=\{\{y_0{:}\check{s}\}\}$
$\mathcal{L}^H_\circ(4)=\{\{y_0{:}\check{s}\}\}$
$\mathcal{L}^H_\bullet(4)=\{\{y_0{:}\check{s}, y{:}\tilde{p}\}\}$
$\mathcal{L}^H_\circ(5)=\{\{y_0{:}\check{s}, y{:}\tilde{p}\}\}$
$\mathcal{L}^H_\bullet(5)=\{\{y_0{:}\check{s}, y{:}\tilde{p}\}\}$
$\mathcal{L}^H_\circ(6)=\{\{y_0{:}\check{s}\}\}$
$\mathcal{L}^H_\bullet(6)=\{\{y_0{:}\check{s}, z{:}\tilde{q}\}\}$
$\mathcal{L}^H_\circ(7) =\{\{y_0{:}\check{s}, z{:}\tilde{q}\}\}$
$\mathcal{L}^H_\bullet(7) =\{\{y_0{:}\check{s}, z{:}\tilde{q}\}\}$
$\mathcal{L}^H_\circ(8) =\{\{y_0{:}\check{s}, y{:}\tilde{p}\}, \{y_0{:}\check{s}, z{:}\tilde{q}\}\}$
$\mathcal{L}^H_\bullet(8) =\{\{y_0{:}\check{s}, y{:}\tilde{p}\}, \{y_0{:}\check{s}, z{:}\tilde{q}\}\}$
$\mathcal{L}^H_\circ(9) =\{\emptyset\}$
$\mathcal{L}^H_\bullet(9) =\{\{w{:}\tilde{r}\}$
$\mathcal{L}^H_\circ(10)=\{\{w{:}\tilde{r}\}\}$
$\mathcal{L}^H_\bullet(10)=\{\{w{:}\tilde{r}\}\}$
$\mathcal{L}^H_\circ(11)=\{\{y_0{:}\check{s}, y{:}\tilde{p}\}, \{y_0{:}\check{s}, z{:}\tilde{q}\}, \{w{:}\tilde{r}\}\}$
$\mathcal{L}^H_\bullet(11)=\{\{y_0{:}\tilde{s}, y{:}\tilde{p}\}, \{y_0{:}\tilde{s}, z{:}\tilde{q}\}, \{w{:}\tilde{r}\}\}$
$\mathcal{L}^H_\circ(12)=\{\{y_0{:}\tilde{s}, y{:}\tilde{p}\}, \{y_0{:}\tilde{s}, z{:}\tilde{q}\}, \{w{:}\tilde{r}\}\}$

The scenario at the end of the branch L4–L5 is $\mathcal{L}^H_\bullet(5) = \{\{y_0{:}\check{s}, y{:}\tilde{p}\}\}$, where s is pending and p active. Similarly, at the end of the branch L6–L7 the scenario is $\mathcal{L}^H_\bullet(7) = \{\{y_0{:}\check{s}, z{:}\tilde{q}\}\}$. To compute the state after the inner conditional expression the analysis computes the upper bound of both branches—recall that $\sqcup$ is defined as $\cup$—so $\mathcal{L}^H_\circ(8) = \{\{y_0{:}\check{s}, y{:}\tilde{p}\}, \{y_0{:}\check{s}, z{:}\tilde{q}\}\}$. Finally, to compute the state after the external conditional expression the analysis computes the upper bound of the states after branch L2–L8 and branch L9–L10, resulting in $\mathcal{L}^H_\circ(11) = \mathcal{L}^H_\bullet(8) \sqcup \mathcal{L}^H_\bullet(10) = \{\{y_0{:}\check{s}, y{:}\tilde{p}\}, \{y_0{:}\check{s}, z{:}\tilde{q}\}, \{w{:}\tilde{r}\}\}$. Note that this state contains the information from the 3 possible paths.

In method m2 there is a conditional statement inside a **while** loop, where future variables are omitted for clarity. In this case the analysis computes the upper bound of the states after L13 and L21 in order to obtain the state before L14, i.e., $\mathcal{L}^H_\circ(14) = \mathcal{L}^H_\bullet(13) \sqcup \mathcal{L}^H_\bullet(21)$. When solving iteratively the set of equations it generates every different combination of paths among iterations: $\{\emptyset\}$, $\{\{\star{:}\check{p}\}, \{\star{:}\tilde{q}\}\}$, $\{\{(\star{:}\check{p})^+\}, \{\star{:}\check{p}, \star{:}\tilde{q}\}, \{(\star{:}\tilde{q})^+\}\}$, etc. After some iterations the process converges with the following results:

$$
\begin{array}{rcl}
\mathcal{L}^H_\circ(13) & = & \{\emptyset\} \\
\mathcal{L}^H_\bullet(13) & = & \{\emptyset\} \\
\mathcal{L}^H_\circ(14) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(14) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(15) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(15) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(16) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(16) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(17) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(17) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(18) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(18) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(19) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(19) & = & \{\{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(20) & = & \{\{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(20) & = & \{\{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(21) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(21) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(22) & = & \{\{(\star{:}\check{p})^+\}, \{(\star{:}\check{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\check{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\bullet(22) & = & \{\{(\star{:}\tilde{p})^+\}, \{(\star{:}\tilde{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\tilde{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\tilde{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\} \\
\mathcal{L}^H_\circ(23) & = & \{\{(\star{:}\tilde{p})^+\}, \{(\star{:}\tilde{p})^+, \star{:}\tilde{q}\}, \{(\star{:}\tilde{p})^+, (\star{:}\tilde{q})^+\}, \{\star{:}\tilde{p}, (\star{:}\tilde{q})^+\}, \{(\star{:}\tilde{q})^+\}\}
\end{array}
$$

As can be seen, the process converges when $\mathcal{L}^H_\circ(14) = \mathcal{L}^H_\bullet(21)$, i.e., when the state at the end of the loop is the same as the state at its entry. Note that both branches of the **if** expression (L17–L18 and L19–L20) are combined in $\mathcal{L}^H_\circ(21)$ using the $\sqcup$ operator (set union). In the end, $\mathcal{L}^H_\circ(14)$ collects all possible execution paths inside the loop: a) $\{(\star{:}\check{p})^+\}$, the branch L17–L18 of the **if** expression is always chosen; b) $\{(\star{:}\tilde{q})^+\}$, the branch L19–L20 is always chosen; or c) $\{(\star{:}\check{p})^+, \star{:}\tilde{q}\}$, $\{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}$, $\{\star{:}\check{p}, (\star{:}\tilde{q})^+\}$), any interleaving of branches. Note that abstract states H are sets, so applying the transfer function τ_H can produce states with fewer elements. This situation happens with $\mathcal{L}^H_\bullet(17)$: it is obtained applying τ_H to $\mathcal{L}^H_\circ(17)$ but $\mathcal{L}^H_\bullet(17) \subsetneq \mathcal{L}^H_\circ(17)$ because $\tau(\textbf{this}.\mathsf{p}(), \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\}) = \{(\star{:}\check{p})^+, (\star{:}\tilde{q})^+\} = \tau(\textbf{this}.\mathsf{p}(), \{\star{:}\check{p}, (\star{:}\tilde{q})^+\})$—similar with $\mathcal{L}^H_\bullet(19)$ and $\mathcal{L}^H_\circ(19)$.

It is important to note that computing the method-level information in loops with nested conditional expressions can produce an exponential number of execution paths.

Since an abstract state $H = \{\overline{M_i}\}$ stores the information of the i different execution paths, this growth can make impracticable the method level. In order to mitigate this issue, the actual implementation uses a *threshold* value of execution paths to consider when analyzing a loop. If that value is exceeded, the analysis merges all the simple states $M_i \in H$ into a unitary state $\{M^*\}$ where $M^* = \sqcup_{M_i \in H} M_i$—being $\sqcup$ the upper bound operator defined in Section 4.1.

5.2. Global Analysis

In this section we describe how to modify the construction of the MHP graph, and the corresponding set of MHP pairs, to handle the conditional statements. The MHP graph is similar to the one described in Section 4.2, but instead of *future variable nodes* it contains *path nodes*. Recall that future variables nodes were used to improve the precision when methods calls, that use the same future variable, are performed in different branches of an **if** statement. The role of *path nodes* is similar, it indicates which tasks occur in the same execution path and which are in exclusive execution paths, independently from the future variable they synchronize with. For the rest of this section we assume that the method-level MHP equations $\mathcal{L}_P^H$ has been generated and solved.

Definition 5.6. The (conditional) MHP graph of P is a directed weighted graph $\mathcal{G}_P^H = \langle V, E \rangle$ with a set of nodes V and a set of edges $E = E_1 \cup E_2 \cup E_3$ defined as follows:

$$
\begin{aligned}
V &= \{\tilde{m}, \hat{m}, \check{m} \mid m \in P_{\mathcal{M}}\} \cup P_{\mathcal{P}} \cup \{p_i \mid p \in P_{\mathcal{P}}, \mathcal{L}_\circ^H(p) = \{M_1, \ldots, M_k\}, i \in [1..k]\} \\
E_1 &= \{\tilde{m} \to p \mid m \in P_{\mathcal{M}}, p \in m\} \cup \{\hat{m} \to p_{\check{m}}, \check{m} \to p_{\hat{m}} \mid m \in P_{\mathcal{M}}\} \\
E_2 &= \{p \to p_i \mid p \in P_{\mathcal{P}}, \mathcal{L}_\circ^H(p) = \{M_1, \ldots, M_k\}, i \in [1..k]\} \\
E_3 &= \{p_i \to x \mid p \in P_{\mathcal{P}}, \mathcal{L}_\circ^H(p) = \{M_1, \ldots, M_k\}, y{:}x \in M_i\} \cup \\
&\quad \{p_i \overset{\infty}{\to} x \mid p \in P_{\mathcal{P}}, \mathcal{L}_\circ^H(p) = \{M_1, \ldots, M_k\}, (\star{:}x)^+ \in M_i\}
\end{aligned}
$$

Let us explain the different components of $\mathcal{G}_P^H$. The set of nodes V consists of three kind of nodes: the first two, that correspond to the first two disjuncts, are the same as in the previous MHP graph and they represent methods and program points; the third kind, that corresponds to the third disjunct, is new and it represents execution paths and thus we refer to them as *path nodes*. These kind of nodes is generated using the information in the (conditional) abstract state of each program points p: if $\mathcal{L}_\circ^H(p) = \{M_1, \ldots, M_k\}$, then the MHP graph will contain k path nodes $p_1, \ldots, p_k$. These nodes will be used to avoid generating indirect MHP pairs using exclusive execution paths.

The set of edges E consists of three kinds of edges: E_1 is the same as in the previous MHP graph, it connects method nodes to their corresponding program point nodes; E_2 connects each program point node to its path nodes; and E_3 connects each path node p_i to the method nodes depending on the MHP atoms that correspond to p_i.

Example 5.7. The conditional MHP graph of the program of Figure 8, for some program points of interest, is depicted in the same figure on the right. Note, for example, that program point node 8 is connected to two path nodes 8_1 and 8_2 that correspond to the sets of $\mathcal{L}_\circ^H(8) = \{\{y_0{:}\tilde{s}, y{:}\tilde{p}, z{:}\tilde{q}\}, \{y_0{:}\tilde{s}, w{:}\tilde{r}\}\}$ (see Example 5.2). In turn, path node 8_1 is connected to $\tilde{s}$, $\tilde{p}$ and $\tilde{q}$, and path node 8_2 is connected to $\tilde{s}$ and $\tilde{r}$.

Using the new kind of graphs we can compute the MHP pairs in a similar way to that of Section 4.2, however, with a special treatment for path nodes.

	1	2	5	7	8	9	10	11	13	15	17
1											
2								•			
5								•	•	•	
7								•			•
8								•	•	•	•
9								•	•	•	•
10								•	•	•	•
11		•	•	•	•	•	•		○	○	○
13			•		•	•	•	○		○	○
15			•		•	•	•	○	○		○
17				•	•	•	•	○	○	○	

	1	2	5	7	8	9	10	11	13	15	17
1											
2								•			
5								•	•	•	
7								•			•
8								•	•	•	•
9								•	•	•	•
10								•	•	•	•
11		•	•	•	•	•	•		○	○	○
13			•		•	•	•	○		○	
15			•		•	•	•	○	○		
17				•	•	•	•	○			

Fig. 11. MHP pairs for the program of Figure 8: (left) $\tilde{\mathcal{E}}_P$, using the analysis of Section 4; (right) $\tilde{\mathcal{E}}_P^H$, using the extended analysis of Section 5.

Definition 5.8. The MHP information induced by the (conditional) MHP graph $\mathcal{G}_P^H$ is defined as $\tilde{\mathcal{E}}_P^H = \mathit{directMHP}_c \cup \mathit{indirectMHP}_c$ where

$$\begin{aligned}
\mathit{directMHP}_c = \quad & \{(p_1,p_2),(p_2,p_1) \mid p_1,p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_P^H)\} \\
\mathit{indirectMHP}_c = \cup \; & \{(p_1,p_2) \mid p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \to x_1 \to x_2 \rightsquigarrow p_1 \in \mathcal{G}_P^H, \\
& \qquad p_3 \to x_1 \to x_3 \rightsquigarrow p_2 \in \mathcal{G}_P^H, x_2 \neq x_3\} \\
& \{(p_1,p_2) \mid p_1,p_2,p_3 \in P_{\mathcal{P}}, p_3 \to x_1 \overset{\infty}{\to} x_2 \rightsquigarrow p_1 \in \mathcal{G}_P^H, \\
& \qquad p_3 \to x_1 \overset{\infty}{\to} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P^H\}
\end{aligned}$$

The definition of direct MHP pairs remains the same as in Definition 4.14, but the definition of indirect MHP pairs takes path nodes into account: we say that program points p_1 and p_2 may happen (indirectly) in parallel if they have a common ancestor p_3, which is a program point node, such that the paths $p_3 \rightsquigarrow p_1$ and $p_3 \rightsquigarrow p_2$ visit *the same path node* x_1 first, and then each continues in a different path. Note that this excludes cases in which the paths to p_1 and p_2 involve exclusive execution branches.

Example 5.9. Figure 11 includes the MHP sets $\tilde{\mathcal{E}}_P$ and $\tilde{\mathcal{E}}_P^H$, for some program points of interest, computed using the analysis of Section 4 and the one of this section respectively. The later is computed using the MHP graph of Figure 8. We can see that the direct MHP pairs (marked by •) are the same, however, the indirect MHP pairs (marked by ○) are different. In particular, $\tilde{\mathcal{E}}_P^H$ does not include the pairs $(13, 17)$ and $(15, 17)$ while $\tilde{\mathcal{E}}_P$ does. This is because that paths from the common ancestors, for example L10, go to L13 and L17 through different path nodes. Recall that these pairs correspond to a spurious scenario in which methods p and q can run in parallel with r.

The next theorems state the soundness of the analysis, and compare its precision to that of Section 4. The proofs can be found in the appendix.

THEOREM 5.10 (SOUNDNESS OF $\tilde{\mathcal{E}}_P^H$). $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P^H$

THEOREM 5.11. $\tilde{\mathcal{E}}_P^H \subseteq \tilde{\mathcal{E}}_P$

As regards complexity, the analysis of this section has a higher cost than the one of Section 4, because methods might have an exponential number of execution paths. If a method m has b_m possible execution paths, in the worst case the cost of solving its method-level MHP data-flow equations is in $O(b_m \cdot d \cdot pp_m \cdot nm_m \cdot \mathit{fut}_m \cdot)$, i.e., b_m times the cost of the method-level analysis in Section 4.1—in the worst case it will need to compute an abstract state M for every execution path. However, as mentioned

in Section 5.1, in order to avoid this exponential explosion the implementation uses a threshold value to stop considering the different paths separately and merging them. Computing the MHP pairs from the graph $\mathcal{G}_P^H$ is still a reachability problem that can be solved in $O(n^3)$ where n is the number of nodes, but the new graph typically has more nodes than the one of Section 4.2.

6. APPLICATIONS OF MHP ANALYSIS

In this section, we discuss some applications of our analysis, some of them are being investigated in recent work [Flores-Montoya et al. 2013; Albert et al. 2013].

6.1. Data Race Detection

MHP analysis is a crucial analysis in the context of concurrent and distributed programming. Its most common application is detection of data races. If two instructions that may happen in parallel access the same (global) data, this causes a data race. As a consequence, such data might get inconsistent values, leading to different types of errors (including exceptions, runtime errors, etc.). The ABS language has been designed to be data race free, as fields can only be directly accessed by the **this** object. However, our analysis does not use this restriction and our results are valid as well if accessing fields of other objects is allowed. Therefore, for other asynchronous language that do not have this restriction (e.g., [Emmi et al. 2012]), our analysis can be used to detect data races.

6.2. Deadlock Analysis

Deadlock situations are produced when a concurrent program reaches a state in which one or more tasks are waiting for each other termination and none of them can make any progress. In the concurrent objects paradigm, the combination of non-blocking and blocking mechanisms to access futures may give rise to complex deadlock situations and a rigorous formal analysis is required to ensure deadlock freeness. [Flores-Montoya et al. 2013] proposes an analysis based on constructing a *dependency graph* which, if acyclic, guarantees that the program is deadlock free. However, without *temporal* information, dependency graphs would be extremely imprecise. The crux of this deadlock analysis is the use of our MHP analysis which allows identifying whether the dependencies between the synchronization instructions can happen in parallel. Essentially, the dependency graph is labeled with the program points of the synchronization instructions that introduce the dependencies and, thus, that may potentially induce deadlocks. In a post-process, *unfeasible* cycles in which the synchronization instructions involved in the circular dependency may not happen in parallel are discarded.

Example 6.1. Consider the example in Figure 12 borrowed from [Flores-Montoya et al. 2013]. After creating the Server object a, method go starts to execute. The object a remains blocked in instruction 10 until the execution of go completes. Once it finishes, it invokes acc on the Client object c. The execution of acc makes a call to rec on the server object and then blocks its execution until it completes. By only looking at the dependencies between objects, a deadlock analyzer might report a false deadlock, as object a will be blocked waiting for an answer from c and vice versa. However, there is no deadlock in the execution of main since it is guaranteed that the execution of acc in L11 will start only after the execution of go at L9 has finished. In particular, when the execution of acc blocks the c object at L21 waiting for termination of rec it is guaranteed that a is no longer blocked. The inference of this information requires the enhancement of the dependencies used in a deadlock analysis with MHP relations, namely our analysis provides the information that (10,21) is not an MHP pair. This information allows proving that the program is deadlock free.

```
1 main() {
2  a=new Server();
3  a.go();
4 }
```

```
5 class Server {
6  Unit rec(Str msg) {}
7  Unit go() {
8   c=new Client();
9   y=c.go(this);
10  a=y.get;
11  c.acc();
12 }
13 }
```

```
14 class Client{
15  Server srv;
16  Unit go(Server s) {
17   srv=s;
18  }
19  Unit acc() {
20   y=srv.rec("...");
21   a=y.get;
22  }
23 }
```

Fig. 12. Application of MHP in deadlock analysis

```
24 class Loops (Int field) {
25  Unit loop1() {
26   while ( field > 0 )  {
27    y=this.g();
28    await y?;
29    field--;
30   }
31 }
32 Unit loop2(Int m) {
33  while ( m > 0 )  {
34   y=this.g();
35   await y?;
36   field=*;
37   m--;
38  }
39 }
```

```
40 Unit loop3() {
41  while ( field < 0 )  {
42   y=this.g();
43   await y?
44   field++;
45  }
46 }
47  Unit g() {}
48 } //end class Loops
49 main(Int f, Int m) {
50  a=new Loops(f);
51  y=a.loop1();
52  z=a.loop2(m);
53  await y?
54  w=a.loop3();
55 }
```

Fig. 13. Application of MHP in termination/cost analysis

6.3. Termination Analysis

Termination analysis of concurrent and distributed systems is receiving considerable attention [Popeea and Rybalchenko 2012; Cook et al. 2007]. The main challenge is in handling *shared-memory* concurrent programs. This is because, when execution interleaves from one task to another, the shared-memory may be modified by the interleaved task. The modifications will affect the behavior of the program and, in particular, can change its termination behavior and its resource consumption. [Albert et al. 2013] presents a termination analysis for concurrent objects which assumes a *property* on the global state in order to prove termination of a loop and, then, proves that this property holds. The property to prove is the *finiteness* of the shared-data involved in the termination proof, i.e., proving that such shared-memory is updated a finite number of times. The main idea is that if a loop S terminates under the assumption that a set of fields F are not modified at the release points of S, then S also terminates if they are modified a finite number of times. The intuition is that if the fields are modified finitely, then we will eventually reach a state from which that state on they are not modified. From that state, we cannot have non-termination since we know that S terminates if the fields are not modified. Crucial for accuracy is the use of the information inferred by our MHP analysis which allows restricting the set of program points on which the property has to be proved to those that may actually interleave its execution with the considered loop.

Example 6.2. Consider the example in Figure 13 which is inspired by the examples in [Albert et al. 2013]. After creating the Loops object, method main makes two asyn-

chronous calls to methods loop1 and loop2. The main observation is that the execution of these two loops might interleave since both loops have a release point. Here field is a shared variable and m a local variable. We have that (a) loop1 terminates under the assumption that f does not change at the release point (L28); and (b) loop2 terminates without any assumption, as its counter is a local variable. Now, since loop2 terminates, we know that field is modified a finite number of times at the release point of loop1 and thus loop1 terminates when running in parallel with loop2. However, the interleaved execution of loop1 and loop3 can lead to non-termination, as one loop increases the counter and the other one decreases it. The crucial observation is that our MHP analysis tells us that they cannot execute in parallel. Due to the use of **await** at L53, we know that the execution of loop1 cannot happen in parallel with the execution of loop3, thus we can guarantee termination of the whole program.

6.4. Resource Analysis

Besides termination, the same style of reasoning can be applied to infer the resource consumption (or cost) of executing the concurrent program. The results of the termination analysis already provide useful information for cost: if the program is terminating, we know that the size of all data is bounded. Thus, we can give cost bounds in terms of the maximum and/or minimum values that the involved data can reach. Besides, one needs techniques to infer upper bounds on the number of iterations of loops whose execution might interleave with instructions that update the shared memory. The approach in [Albert et al. 2013] is based on the combination of *local* ranking functions (i.e., ranking functions obtained by ignoring the concurrent interleaving behaviors) with upper bounds on the *number of visits* to the instructions which update the shared memory. As in the case of the termination analysis, our MHP analysis is used to restrict the set of points whose visits have to be counted to those that indeed may interleave. For instance, when obtaining an upper bound on the number of iterations of loop1 in the above example, we do not need to count the number of visits to the instruction 44 in loop3, as it cannot happen in parallel.

Besides, the MHP analysis has been crucial to infer the peak cost of distributed systems [Albert et al. 2014b]. The peak cost of a distributed location refers to the maximum cost that its locations need to carry out along its execution. The notion of peak relies on the concept of *quantified queue configuration* which captures the worst-case cost of the tasks that may be simultaneously pending to execute at each location along the execution. A particular queue configuration is given as the sets of tasks that the location may have pending to execute at a moment of time. This information is provided by our may-happen-in-parallel analysis.

7. EXPERIMENTAL EVALUATION

We have implemented our analysis as a system called MayPar that can be tried out independently online at: `http://costa.ls.fi.upm.es/costabs/mhp`. Besides, it is integrated within the SACO system [Albert et al. 2014a] where it is used to prove termination, deadlock and obtain bounds on the resource consumption. 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 small concurrent applications: PeerToPeer, a peer to peer protocol implementation; Chat, a client-server implementation of a chat program; BookShop, a web shop client-server application; BBuffer, a classical bounded-buffer for communicating several producers and consumers; DistHT, a distributed hash-table; and MailServer, a simple model of a mail server.

All experiments have been performed on an Intel Core i7-3667U at 2.00GHzx4 with 8GB of RAM, running Ubuntu 12.04. Table 14 summarizes experiments with the basic

Code	MHP						Peak cost	
	$\mathbf{E_p}$	$\tilde{\mathcal{E}}_\mathbf{P}$	$\mathbf{PPs^2}$	$\mathbf{R}\varepsilon$	$\mathbf{T_G}$	$\mathbf{T}_{\tilde{\mathcal{E}}_\mathbf{P}}$	$\#_c$	Gain
ReplicationSystem	-	68925	87025	-	146	3011	-	-
TradingSystem	-	14829	18769	-	70	420	-	-
PeerToPeer	385	487	1296	7.87%	<10	18	9	0.93
Chat	552	1287	2209	33.27%	<10	30	10	0.07
BookShop	66	66	196	0%	<10	<10	6	0.20
BBuffer	36	36	49	0%	<10	<10	6	0.97
DistHT	83	151	576	12%	<10	<10	4	0.22
MailServer	17	34	64	26.5%	<10	<10	6	0.72

Fig. 14. Basic Analysis (times are in milliseconds)

analysis. [1] For each program, $\mathcal{G}_P$ is built and the relation $\tilde{\mathcal{E}}_P$ is completely computed using only the program points required for soundness (see Section 4.5). $\mathbf{E_p}$ 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 (10000 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. Such measure does not include the time required for printing the pairs into the standard output.

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. In the small examples, the analyzer achieves high precision, with the approximated error less than 33.27% (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, the biggest case study (ReplicationSystem) required 3157 milliseconds. Besides, although in the experiments we have computed the complete set of MHP pairs, for most applications, only a reduced set of points needs to be queried. In conclusion, we argue that our experiments prove that our analysis is both accurate and efficient in practice.

The extension of the analysis that improves the precision when handling conditional expressions presented in Section 5 has not been integrated into the MayPar system. The reason is that, as the MHP analysis is an essential part of other analysis (see Section 6), we have preferred to use the lighter and simpler version presented in Section 4 for pragmatic reasons. However we have performed an evaluation of the impact it would have in the benchmark programs used in this section. The evaluation was done by inspecting the code looking for conditional statements and detecting the

[1] The parameters used for these experiments are *"Standard MHP"*, *"Compute MHP pairs"*, *"Ignore exit program points when computing the MHP pairs"*, *"Verbosity: 1"* and *"Type: Reduced"*.

potential gain of precision at those points. For the set of small concurrent programs (PeerToPeer, Chat, BookShop, BBuffer, DistHT and MailServer) we conclude that the extended analysis do not achieve any precision improvement. The reason is that these programs contain very few conditional expressions, and they do not invoke tasks in both their **then** and **else** branches. On the other hand, the extended analysis would obtain precision improvements for the industrial case studies ReplicationSystem and TradingSystem. These programs contain a higher number of conditional expressions nested in complex ways, thus producing different paths of execution containing different task invocations. Moreover, we have detected a pattern that appears several times in these programs, like the following fragment from ReplicationSystem (recall that the ABS language allows calling methods without association to future variables):

```
1 if (command == SkipFile) {
2     ...
3 } else if (command == OverwriteFile) {
4     this.overwrite(file);
5 } else if (command == ContinueFile) {
6     this.continue(file);
7 }
```

In these cases the program invokes different tasks depending on the evaluation of some parameter. The original analysis from Section 4 would infer that tasks overwrite and continue can happen in parallel because the calls do not share the same future variable, therefore generating MHP pairs between the program points of these two tasks and also between any task that is directly or indirectly invoked by them. Since the extended analysis presented in Section 5 keeps separated the status of both invocations because they belong to different paths of execution, it would infer that overwrite cannot happen in parallel with continue, thus avoiding a number of spurious MHP pairs.

As mentioned in Section 6, the MHP analysis has a great impact on other static analysis like termination or resource analysis. In particular, MHP analysis is crucial to infer the peak cost of distributed systems [Albert et al. 2014b], i.e., the maximum cost that its locations (distributed nodes) need to carry out along its execution. In our language, the locations are the objects. As explained in Section 6, the analysis infers the queue configuration for each object that (over)approximates the tasks that can be simultaneously in the object queue. To show this impact, Table 14 contains two columns under the heading *"Peak cost"*: $\#_c$ is the number of locations created during the execution of the program, and *Gain* is the precision gain when using the MHP information. In order to measure this gain we have computed the peak cost in each location using the MHP pairs obtained by the MHP analysis ($peak_{MHP}$) and without using them ($peak$), and calculated the proportion $peak_{MHP}/peak$. Note that $peak$ accumulates the total cost in a location along the whole program execution, since there is no way to distinguish which tasks cannot happen in parallel without the MHP information. As the peak cost is a complex expression whose variables are the input parameters of the main method, we have obtained the average gain for each location by evaluating the cost expressions for 15 random values. For each program, the column *Gain* contains the minimum proportion, i.e., the maximum gain, obtained among the locations.

Table 14 shows that although the gain on peak cost ranges from 0.07 to 0.97, the results are good in general, with and average maximum gain of 0.52. The programs ReplicationSystem and TradingSystem are not considered in this comparison because it has not been possible to compute their peak cost expressions due to limitations in the peak cost analysis. Thus, there is no measure of gain in these cases. The gain obtained in BBuffer and PeerToPeer is the lowest as those programs have a lot of parallelism and the MHP information can only reduce a few queue configurations. Similarly MailServer

has a moderate maximum gain of 0.72. On the other hand Chat, BookShop and DistHT obtain greater gains because their synchronization points produce fewer and smaller queue configurations (as tasks are awaited for termination and hence MHP pairs are discarded), so the peak cost analysis obtains more precise cost expressions. Notice that it is not possible to infer a relation between the precision of the MHP (the approximated error percentage $\mathbf{R}\varepsilon$) and the gain obtained. There are two reasons for this fact: First, the error percentage of the MHP analysis is low in general, with a maximum of 33.27%; and second, the peak cost gain is more related to the overall parallelism of a program than to the precision of the MHP analysis. A program with a very high level of parallelism, where many tasks are executed in objects at the same time, will have a discreet gain in the peak cost even if the MHP analysis obtains the actual MHP pairs. On the other extreme, programs like Chat—with a moderate error percentage of 33.27%—obtains a great gain of 0.07. The precision of the MHP analysis is not as high as in other examples, however the Chat program has an intrinsically low level of parallelism (few tasks are executed in objects at the same time). Therefore the MHP information inferred by the analysis, although could be improved, is enough to discard and prune many queue configurations, resulting in a very high gain in the peak cost.

8. RELATED WORK

Some of the earlier works on MHP analysis are for Ada programs [Duesterwald and Soffa 1991; Masticola and Ryder 1993; Naumovich and Avrunin 1998]. Ada's rendezvous concurrency model is based on synchronous message passing which is very different to the use future variables. In addition, the analyses of [Masticola and Ryder 1993; Naumovich and Avrunin 1998] assume that the total number of tasks is limited and can be known at compile time. This limitation is relaxed in [Masticola 1993] where the analysis of [Masticola and Ryder 1993] is adapted to concurrent C.

The approach of [Naumovich and Avrunin 1998] was extended to Java programs in [Naumovich et al. 1999]. It consists on a data flow analysis over a so-called *Parallel Execution Graph* (PEG) which is the union of all the control flow graphs of each thread with additional edges at the synchronization points. This analysis support `start` and `join` for creating threads and waiting for them to finalize; and `wait` and `notify` for synchronization within monitors. The primitives `start` and `join` are very similar our method call and **await** $y?$. However, this approach has some limitations that make it inappropriate for our concurrency model. Because all the threads are explicitly represented in the PEG, the analysis cannot deal with programs with unbounded thread creation. Besides, it requires the inlining of the procedures that can intervene in the synchronization. The analysis in [Naumovich et al. 1999] is improved in [Li and Verbrugge 2004] to achieve a better efficiency in the implementation but the mentioned limitations remain.

This limitation is overcome by [Barik 2005]. The analysis of [Barik 2005] can deal with unbounded thread creation and termination (`start` and `join`) and locks but not with `wait` and `notify`. The work of [Barik 2005] is in fact the closest to ours in both the concurrency model and the techniques used. In [Barik 2005], Java programs are abstracted to a *thread creation tree* (TCT) in which each node is an abstract thread (that might represent several concrete threads) obtained using symbolic execution. These TCTs play a similar role of our MHP graphs where there are only method nodes (although our graphs are not necessarily trees). The MHP inference in [Barik 2005] has two phases. The first phase infers a coarse-grained MHP information at the level of threads with reachability conditions similar to our *directMHP* and *indirectMHP*. In the second phase, the direct MHP information is refined taking the internal structure of the abstract threads into account. In our approach, the internal structure of the

methods is considered in the local analysis and reflected in the MHP graph which contains all the relevant information.

The fact that we deal with a language where concurrency constructs and objects are integrated, allows us to have a more modular and incremental analysis. In our analysis the method-level analysis can be performed independently for each method, whereas in [Barik 2005] a symbolic execution of the whole program is necessary. Another important difference with respect to [Barik 2005] is that we define the MHP property with respect to the semantics of the language and prove its correctness whereas [Barik 2005] provides only an informal definition.

More recently, several MHP analyses for X10 have been proposed [Agarwal et al. 2007; Lee and Palsberg 2010; Lee et al. 2012]. X10 has async-finish parallelism which differs from ours substantially. This kind of structured parallelism 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. The analysis in [Agarwal et al. 2007] computes the *Never-execute-in-Parallel*, the complement of the MHP property. The analysis in only intraprocedural and it is based on checking certain conditions on the *Program structure Tree* (PST) which is a reduced version of a procedure abstract syntax tree. Afterwards, the results are refined by taking places and atomic sections into account.

Finally, both [Lee and Palsberg 2010; Lee et al. 2012] present a MHP analysis of X10 based on a type system. They generate set constraints from the types and they infer the MHP information by solving the constraints. They provide formal semantics of featherweight X10 (a reduced version of X10) and the prove the correctness of their analysis with respect to the semantics. Moreover, in [Lee et al. 2012], the authors prove that the result is precise for non-recursive programs with respect to some storeless semantics (where loops and conditionals are non-deterministic). They also provide a method for obtaining a precise result for recursive programs using *Constrained Dynamic Pushdown Networks* (CDPNs).

As we have seen in Section 6, other analyses for more complex properties can greatly benefit from the MHP pairs that our MHP analysis infers. Several proposals for deadlock analysis [Naik et al. 2009; Flores-Montoya et al. 2013] rely on the MHP pairs to discard unfeasible deadlocks when the instructions involved in a possible deadlock cycle cannot happen in parallel. In a more recent application, [Albert et al. 2013], the MHP analysis also plays a fundamental role to increase the accuracy of termination and resource analysis.

Additionally, the analysis can be made more precise by making it *object-sensitive* using standard techniques such as those in [Whaley and Lam 2004; Milanova et al. 2002]. Object sensitive analysis allows us to infer whether 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 [Whaley and Lam 2004; Milanova et al. 2002].

9. CONCLUSIONS AND FUTURE WORK

We have proposed a novel and efficient approach to infer MHP information for actor-based programs. This MHP information is essential to infer more complex properties of concurrent programs, namely the precision of other static analysis like deadlock, termination and resource analysis is greatly increased. The main novelty of our analysis 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. In addition, we have defined the necessary conditions to perform a partial analysis based on a set of program points of interest (Section 4.5) and we have developed an extension to the basic analysis that increases the accuracy in the presence of conditional expressions (Section 5).

We are currently working on the extension of our analysis to *inter-procedural synchronization*, i.e., we can synchronize with the termination of a task outside the scope in which the task is spawned, as it is available in many concurrent languages. The enhancement to inter-procedural synchronization requires the development of a *must-have-finished* analysis which infers inter-procedural dependencies among the tasks. Such dependencies will allow us to determine that, when a task finishes, those that are awaited for on it must have finished as well. We refer to [Albert et al. 2015] for a detailed description of this extension and of its integration with the analysis presented in this article. In future work, we plan to further investigate novel applications of our analysis. We are interested in taking the parallelism into account in order to statically predict the execution time of a parallel system accurately. Our objective is to be able to infer the fragments of code that execute in parallel and then take the maximum execution time of this parallel fragments.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful suggestions.

REFERENCES

Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. 2007. May-happen-in-parallel Analysis of X10 Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, New York, NY, USA, 183–193. DOI: http://dx.doi.org/10.1145/1229428.1229471

G. Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA.

A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers – Principles, Techniques and Tools*. Addison-Wesley.

Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. 2014a. SACO: Static Analyzer for Concurrent Objects. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014 (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 562–567. DOI: http://dx.doi.org/10.1007/978-3-642-54862-8_46

Elvira Albert, Jesús Correas, and Guillermo Román-Díez. 2014b. Peak Cost Analysis of Distributed Systems. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Markus Müller-Olm and Helmut Seidl (Eds.), Vol. 8723. Springer, 18–33. DOI: http://dx.doi.org/10.1007/978-3-319-10936-7_2

Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. 2012. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings (Lecture Notes in Computer Science)*, Holger Giese and Grigore Rosu (Eds.), Vol. 7273. Springer, 35–51. DOI: http://dx.doi.org/10.1007/978-3-642-30793-5_3

Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. 2013. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings (Lecture Notes in Computer Science)*, Dang Van Hung and Mizuhito Ogawa (Eds.), Vol. 8172. Springer, 349–364. DOI: http://dx.doi.org/10.1007/978-3-319-02444-8_25

Elvira Albert, Samir Genaim, and Pablo Gordillo. 2015. May-Happen-in-Parallel Analysis for Asynchronous Programs with Inter-Procedural Synchronization. In *Static Analysis - 22nd International Symposium, SAS 2015. Proceedings (Lecture Notes in Computer Science)*. Springer. To appear.

Rajkishore Barik. 2005. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. In *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers (Lecture Notes in Computer Science)*, Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan (Eds.), Vol. 4339. Springer, 152–169.

Peter A. Buhr, Michel Fortier, and Michael H. Coffin. 1995. Monitor Classification. *ACM Comput. Surv.* 27, 1 (March 1995), 63–107. DOI: http://dx.doi.org/10.1145/214037.214100

Dave Clarke, Einar Johnsen, and Olaf Owe. 2010. Concurrent Objects à la Carte. In *Concurrency, Compositionality, and Correctness*, Dennis Dams, Ulrich Hannemann, and Martin Steffen (Eds.). Lecture Notes in Computer Science, Vol. 5930. Springer Berlin / Heidelberg, 185–206. http://dx.doi.org/10.1007/978-3-642-11512-7_12

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2007. Proving thread termination. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*. ACM, New York, NY, USA, 320–330. DOI: http://dx.doi.org/10.1145/1250734.1250771

P. Cousot and R. Cousot. 1979. A constructive characterization of the lattices of all retractions, pre-closure, quasi-closure and closure operators on a complete lattice. *Portugaliæ Mathematica* 38, 2 (1979), 185–198.

Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. 2007. A Complete Guide to the Future. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science)*, Rocco de Nicola (Ed.), Vol. 4421. Springer, 316–330.

Evelyn Duesterwald and Mary Lou Soffa. 1991. Concurrency Analysis in the Presence of Procedures Using a Data-flow Framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*. ACM, New York, NY, USA, 36–48. DOI: http://dx.doi.org/10.1145/120807.120811

Michael Emmi, Akash Lal, and Shaz Qadeer. 2012. Asynchronous Programs with Prioritized Task-buffers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 48, 11 pages. DOI: http://dx.doi.org/10.1145/2393596.2393652

Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. 2013. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *Formal Techniques for Distributed Systems (FMOODS/FORTE 2013) (Lecture Notes in Computer Science)*, Dirk Beyer and Michele Boreale (Eds.), Vol. 7892. Springer, 273–288. http://dx.doi.org/10.1007/978-3-642-38592-6_19

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2013. Abstract Interpretation over Non-lattice Abstract Domains. In *Proceedings of the 20th International Symposium on Static Analysis, SAS 2013*, Francesco Logozzo and Manuel Fähndrich (Eds.), Vol. 7935. Springer, 6–24. DOI: http://dx.doi.org/10.1007/978-3-642-38856-9_3

Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.* 410, 2-3 (Feb. 2009), 202–220. DOI: http://dx.doi.org/10.1016/j.tcs.2008.09.019

Matthew S. Hecht and Jeffrey D. Ullman. 1973. Analysis of a Simple Algorithm for Global Data Flow Problems. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. ACM, New York, NY, USA, 207–217. DOI: http://dx.doi.org/10.1145/512927.512946

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects - 9th International Symposium, FMCO 2010, Graz, Austria, November 29 - December 1, 2010. Revised Papers (Lecture Notes in Computer Science)*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.), Vol. 6957. Springer, 142–164.

John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *J. ACM* 23, 1 (Jan. 1976), 158–171. DOI: http://dx.doi.org/10.1145/321921.321938

JonathanK. Lee, Jens Palsberg, Rupak Majumdar, and Hong Hong. 2012. Efficient May Happen in Parallel Analysis for Async-Finish Parallelism. In *Static Analysis*, Antoine Miné and David Schmidt (Eds.). Lecture Notes in Computer Science, Vol. 7460. Springer Berlin Heidelberg, 5–23. DOI: http://dx.doi.org/10.1007/978-3-642-33125-1_4

Jonathan K. Lee and Jens Palsberg. 2010. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, NY, USA, 25–36.

Lin Li and Clark Verbrugge. 2004. A Practical MHP Information Analysis for Concurrent Java Programs. In *Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004,*

West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers (Lecture Notes in Computer Science), Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff (Eds.), Vol. 3602. Springer, 194–208.

Stephen P. Masticola. 1993. *Static Detection Of Deadlocks In Polynomial Time*. Technical Report.

Stephen P. Masticola and Barbara G. Ryder. 1993. Non-concurrency Analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. ACM, New York, NY, USA, 129–138. DOI: http://dx.doi.org/10.1145/155332.155346

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 1–11. DOI: http://dx.doi.org/10.1145/566172.566174

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 386–396. DOI: http://dx.doi.org/10.1109/ICSE.2009.5070538

Gleb Naumovich and George S. Avrunin. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Softw. Eng. Notes* 23, 6 (1998), 24–34. 288213.

Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. 1999. An efficient algorithm for computing *MHP* information for concurrent Java programs. *SIGSOFT Softw. Eng. Notes* 24, 6 (1999), 338–354. 319252.

F. Nielson, H. R. Nielson, and C. Hankin. 2005. *Principles of Program Analysis*. Springer. Second Ed.

Benjamin C. Pierce. 1994. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming, International Workshop TPPP'94, Sendai, Japan, November 7-9, 1994, Proceedings (Lecture Notes in Computer Science)*, Vol. 907. Springer, 187–215.

Corneliu Popeea and Andrey Rybalchenko. 2012. Compositional Termination Proofs for Multi-threaded Programs. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*. Springer-Verlag, Berlin, Heidelberg, 237–251. DOI: http://dx.doi.org/10.1007/978-3-642-28756-5_17

G. Ramalingam. 2000. Context-sensitive Synchronization-sensitive Analysis is Undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 416–430. DOI: http://dx.doi.org/10.1145/349214.349241

Richard N. Taylor. 1983. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica* 19, 1 (1983), 57–84. DOI: http://dx.doi.org/10.1007/BF00263928

John Whaley and Monica S. Lam. 2004. Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. DOI: http://dx.doi.org/10.1145/996841.996859

Online Appendix to: May-Happen-in-Parallel Analysis for Actor-based Concurrency

Elvira Albert, Universidad Complutense de Madrid
Antonio Flores-Montoya, Technische Universität Darmstadt
Samir Genaim, Universidad Complutense de Madrid
Enrique Martin-Martin, Universidad Complutense de Madrid

A. PROOFS

This appendix contains the proofs of the results of the paper. Section A.1 presents some auxiliary notions and results that are used in the rest of proofs, whereas Sections A.2 and A.3 contains the proofs of the results of Sections 4 and 5 respectively.

A.1. Auxiliary notions and results

In order to prove the soundness of the analyses we extend the representation of program states and the corresponding semantics. The modified semantics is shown in Fig. 15. Each task contains additional information $tsk(tid, bid, m, l, s, \mathcal{L}r)$. $\mathcal{L}r$ is a set that records the calls that have been performed by the current task, and their status. It can be seen as a concrete version of $\mathcal{L}_{_P}$. For each call, it contains information about the related future variable if there is one and whether the call might be running $\tilde{tid}$, yet to be started $\check{tid}$ or finished $\hat{tid}$. Because task identifiers tid are unique, the sets $\mathcal{L}r$ do not contain *multiple* atoms and the operations over the sets $\mathcal{L}r$ are expressed as substitutions. The notation $\mathcal{L}r[y{:}x/y'{:}x']$ represents a substitution in the set $\mathcal{L}r$ of the atoms that match $y{:}x$ by the corresponding atoms $y'{:}x'$ and _ matches any future variable or $\star$.

Next we need an auxiliary definition for representing the mhp information in the runtime.

Definition A.1 (*Runtime MHP*). Given a program P, we let $\mathcal{E}^r_P = \cup\{\mathcal{E}^r_S \mid S_0 \rightsquigarrow^* S\}$ where $\mathcal{E}^r_S$ is defined as

$$\mathcal{E}^r_S = \left\{ ((tid_1, pp(s_1)), (tid_2, pp(s_2))) \;\middle|\; \begin{array}{l} tsk(tid_1, \_, \_, \_, s_1, \_), tsk(tid_2, \_, \_, \_, s_2, \_) \in S \\ tid_1 \neq tid_2 \end{array} \right\}.$$

Note that $\mathcal{E}_P$ and $\mathcal{E}_S$ can be directly obtained from $\mathcal{E}^r_P$ and $\mathcal{E}^r_S$.

Definition A.2 (*Concrete MHP graph*). Given a state S, we define a concrete graph $\mathcal{G}r_{_S}$ using $\mathcal{L}r$ as follows

$$\begin{array}{ll}
\mathcal{G}r_{_S} = & \langle V_S, E_S \rangle \\
V_S = & \{\tilde{tid}, \check{tid}, \hat{tid} |\ tsk(tid, bid, m, l, s, \mathcal{L}r) \in S\} \cup cP_{_S} \\
cP_{_S} = & \{(tid, pp(s)) |\ tsk(tid, bid, m, l, s, \mathcal{L}r) \in S\} \\
E_S = & ei_S \cup el_S \\
ei_S = & \{\tilde{tid} \rightarrow (tid, pp(s)) |\ tsk(tid, bid, m, l, s, \mathcal{L}r) \in S\} \\
& \cup \{\hat{tid} \rightarrow (tid, p_{\hat{m}}) |\ tsk(tid, bid, m, l, \epsilon(v), \mathcal{L}r) \in S\} \\
& \cup \{\check{tid} \rightarrow (tid, p_{\hat{m}}) |\ tsk(tid, bid, m, l, s, \mathcal{L}r), obj(bid, f, lk) \in S, s = p_{\hat{m}}, lk \neq tid\} \\
el_S = & \{(tid, pp(s)) \rightarrow x |\ tsk(tid, bid, m, l, s, \mathcal{L}r) \in Tk_S \wedge _{:}x \in \mathcal{L}r\}
\end{array}$$

DOI: http://dx.doi.org/10.1145/0000000.0000000

$$\frac{\mathbf{fresh}(bid')\ ,\ l' = l[a \rightarrow bid'], f' = init\_atts(C, \bar{e})}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle a = \textbf{new}\ C(\bar{e}); s\rangle, \mathcal{L}\mathbf{r}) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l', s, \mathcal{L}\mathbf{r}),\ obj(bid', f', \bot) \end{array}}\ (\textsc{newobject})$$

$$\frac{s \neq \epsilon(v)}{obj(bid, f, \bot),\ tsk(tid, bid, \_, \_, s, \mathcal{L}\mathbf{r}) \rightsquigarrow obj(bid, f, tid),\ tsk(tid, bid, \_, \_, s, \mathcal{L}\mathbf{r})}\ (\textsc{select})$$

$$\frac{\begin{array}{c} l(a) = bid_1,\ \mathbf{fresh}(tid_1),\ l' = l[y \rightarrow tid_1],\ l_1 = buildLocals(\bar{e}, m_1) \\ ((bid_1 \neq bid \wedge \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}x/\star{:}x] + y{:}\tilde{tid}_1 \\ \vee (bid_1 = bid \wedge \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}x/\star{:}x] + y{:}\check{tid}_1 \end{array}}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle y = a.m_1(\bar{e}); s\rangle, \mathcal{L}\mathbf{r}) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l', s, \mathcal{L}\mathbf{r}'),\ tsk(tid_1, bid_1, m_1, l_1, body(m_1), \emptyset) \end{array}}\ (\textsc{async})$$

$$\frac{l(y) = tid_1,\ s_1 = \epsilon(v),\ \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}\tilde{x}/y{:}\hat{x}]}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{await}\ y?; s\rangle, \mathcal{L}\mathbf{r}),\ tsk(tid_1, \_, \_, \_, s_1, \_) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, bid, m, l, s, \mathcal{L}\mathbf{r}'),\ tsk(tid_1, \_, \_, \_, s_1, \_) \end{array}}\ (\textsc{await1})$$

$$\frac{l(y) = tid_1, s_1 \neq \epsilon(v),\ \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[\_{:}\check{x}/\_{:}\tilde{x}]}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{await}\ y?; s\rangle, \mathcal{L}\mathbf{r}),\ tsk(tid_1, \_, \_, s_1, \_) \rightsquigarrow \\ obj(bid, f, \bot),\ \{tsk(tid, bid, m, l, \langle \textbf{await}\ y?; s\rangle, \mathcal{L}\mathbf{r}'),\ tsk(tid_1, \_, \_, s_1, \_) \end{array}}\ (\textsc{await2})$$

$$\frac{l(y) = tid_1, s_1 = \epsilon(v), l' = l[a \rightarrow v],\ \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}\tilde{x}/y{:}\hat{x}]}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle a{=}y.\textbf{get}; s\rangle, \mathcal{L}\mathbf{r}),\ tsk(tid_1, \_, \_, \_, s_1, \_) \rightsquigarrow \\ obj(bid, f, tid),\ tsk(tid, m, bid, l', s, \mathcal{L}\mathbf{r}'),\ tsk(tid_1, \_, \_, \_, s_1, \_) \end{array}}\ (\textsc{get})$$

$$\frac{\mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[\_{:}\check{x}/\_{:}\tilde{x}]}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{release}; s\rangle, \mathcal{L}\mathbf{r}) \rightsquigarrow \\ obj(bid, f, \bot),\ tsk(tid, bid, m, l, s, \mathcal{L}\mathbf{r}') \end{array}}\ (\textsc{release})$$

$$\frac{v = l(a),\ \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[\_{:}\check{x}/\_{:}\tilde{x}]}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle \textbf{return}\ a; s\rangle, \mathcal{L}\mathbf{r}) \rightsquigarrow \\ obj(bid, f, \bot),\ tsk(tid, bid, m, l, \epsilon(v)), \mathcal{L}\mathbf{r}') \end{array}}\ (\textsc{return})$$

$$\frac{\begin{array}{c} (f', l', s') = eval(instr, f \cup l) \\ instr \in \{x{=}e,\ \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ _2, \textbf{while}\ b\ \textbf{do}\ s_3\} \end{array}}{\begin{array}{c} obj(bid, f, tid),\ tsk(tid, bid, m, l, \langle instr; k\rangle, \mathcal{L}\mathbf{r}) \rightsquigarrow \\ obj(bid, f', tid),\ tsk(tid, bid, m, l', \langle s'; k\rangle, \mathcal{L}\mathbf{r}) \end{array}}\ (\textsc{sequential})$$

Fig. 15. Extended Semantics with local information

Once the graph has been constructed, we use it to define our $\mathcal{E}G^r_S$, which is a set of MHP relation induced by this graph.

Definition A.3 (*MHP-Graph relation*).

$$\begin{array}{rl} \mathcal{E}G^r_S = & dMHP_S \cup iMHP_S \\ dMHP_S = & \{(x, y) | x, y \in cP_{\mathcal{S}} \wedge x \rightsquigarrow y\} \\ iMHP_S = & \{(x, y) | x, y \in cP_{\mathcal{S}} \wedge (\exists z \in cP_{\mathcal{S}} : z \rightsquigarrow x \wedge z \rightsquigarrow y)\} \end{array}$$

Next we define a function φ, which allows obtaining, by mean of an abstraction, the set $\mathcal{E}_P$ from $\mathcal{E}^r_P$.

Definition A.4 (*MHP relation abstraction*). Let φ be the abstraction function defined as $\varphi : \mathcal{T} \times P_{\mathcal{P}} \rightarrow P_{\mathcal{P}}$ such that $\varphi(tid_1, p_1) = p_1$. The abstraction of the MHP-Graph

relation $\mathcal{E}G^r_S$ is the abstraction of each of its pairs: $\mathcal{E}G_S = \{(\varphi(tid_1, p_1), \varphi(tid_2, p_2)) \mid (tid_1, p_1, tid_2, p_2) \in \mathcal{E}G^r_S\} = \{(p_1, p_2) \mid (tid_1, p_1, tid_2, p_2) \in \mathcal{E}G^r_S\}$

Definition A.5. ψ abstracts $\mathcal{L}r$ sets into sets in $\mathcal{B}$; ψ' abstracts a single MHP atom; and ψ'' abstracts tasks into methods.

$$\begin{aligned}
\psi''(\tilde{t\mathit{id}}) &= \tilde{m} \\
\psi''(\check{t\mathit{id}}) &= \check{m} \\
\psi''(\hat{t\mathit{id}}) &= \hat{m} \text{ where } m = method(tid) \\
\psi'(y{:}x) &= y{:}\psi''(x) \\
\psi(\mathcal{L}r) &= \{y{:}x \mid \exists a \in \mathcal{L}r :\ \psi'(a) = y{:}x \wedge \forall b \in \mathcal{L}r \wedge b \neq a \rightarrow y{:}x \neq \psi'(b))\} \\
&\quad \cup \{(y{:}x)^+ \mid \exists a, b \in \mathcal{L}r :\ a \neq b \wedge \psi'(a) = \psi'(b) = y{:}x\}
\end{aligned}$$

The next three Lemmas state some important properties needed in the proofs: Lemma A.6 shows that tasks without the object's lock cannot contain pending atoms in their $\mathcal{L}r$, Lemma A.7 states that $\mathcal{L}_{P}$ is an approximation of all possible $\mathcal{L}r$ sets and Lemma A.8 shows that paths in the concrete graph $\mathcal{G}r_{S}$ are preserved.

LEMMA A.6. $\forall S : S_0 \rightsquigarrow^* S : \forall\ tsk(tid, bid, m, l, s, \mathcal{L}r) \in S, obj(bid, f, lk) \in S : (lk \neq tid \rightarrow \nexists y{:}\check{x} \in \mathcal{L}r)$

PROOF. Proof by induction on the applied semantic rules

Base case. The theorem trivially holds for S_0.

Inductive case. We assume the theorem holds in the left side of each rule and see if it holds in its right side. Note that any task that has the object's lock satisfies the property. If a task satisfies the property without the lock before a transition and its $\mathcal{L}r$ is not modified in the transition, it will also satisfy the property after the transition.

— (NEWOBJECT) does not change the values of the locks, it only adds a new object $obj(bid', f', \bot)$ with no tasks inside bid' is fresh.
— (SELECT) Only one task obtains the lock which does not affect the property.
— (ASYNC) The task that executes the call has the lock. The created task does not have the lock but its $\mathcal{L}r = \emptyset$ is empty.
— (AWAIT1), (GET) and (SEQUENTIAL) do not lose the lock.
— (AWAIT2), (RELEASE) and (RETURN) release the lock but at the same time, any $\check{x}$ in its $\mathcal{L}r$ set is substituted by $\tilde{x}$.

□

LEMMA A.7 (SOUNDNESS OF $\mathcal{L}_{P}$). $\forall S : S_i \rightsquigarrow^* S : tsk(tid, bid, m, l, s, \mathcal{L}r) \in S \Rightarrow \psi(\mathcal{L}r) \sqsubseteq \mathcal{L}_\circ(\varphi(tid, pp(s)))$

That is, the computed $\mathcal{L}_{P}$ is a safe approximation of the concrete property defined in the semantics.

PROOF. When a task $tsk(tid, bid, m, l, s, \mathcal{L}r)$ is created, $\mathcal{L}r = \emptyset$, $\psi(\mathcal{L}r) = \emptyset$, and $\mathcal{L}_{P} = (\varphi(tid, pp(s)) = \mathcal{L}_\circ(p_{\check{m}}) = \emptyset$ by definition. The transition function τ, together with $\mathcal{L}_{P}$, is equivalent to the transformations of $\mathcal{L}r$ performed in the semantics:

— Neither (NEWOBJECT), (SEQUENTIAL), (SELECT) or τ applied to the corresponding instructions change $\mathcal{L}r$.
— (ASYNC) corresponds to the cases (1) and (2) in τ;
— (AWAIT1), (GET) correspond to the cases (4) and (5) in τ.

— (RELEASE) and (RETURN) correspond to the cases (3) in τ and (6).
— (AWAIT2) is handled directly when generating $\mathcal{L}_{P}$.
— When branching occurs in rule (SEQUENTIAL) (**if** and **while** statements), the upper bound operation is applied in $\mathcal{L}_{P}$ to obtain a joint state that represents all possible branches.

□

LEMMA A.8 (PATH PRESERVATION).
$\forall S : S_i \rightsquigarrow^* S : \forall x, x_1 \in cP_{\mathcal{S}} : x \overset{\mathcal{G}_{r_S}}{\rightsquigarrow} x_1 \Rightarrow \varphi(x) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(x_1)$. *That is, any path in the concrete MHP graph has a corresponding abstract path in the MHP graph.*

PROOF. We can prove the lemma by induction on the length of the paths. In the base case, we consider a minimal path among two program point nodes $p = ltl_1$ where $l = (tid, pp(s))$, $l_1 = (tid', pp(s'))$ and $t \in \{\tilde{tid}, \check{tid}, \hat{tid}\}$.

If p belongs to $\mathcal{G}_{\mathcal{S}}$ then $\exists\ tsk(tid, bid, m, l, s, \mathcal{L}r) \in S : e{:}t \in \mathcal{L}r$ where e might be a future variable y or $\star$. We know that $at = e{:}\psi''(t) \in \psi(\mathcal{L}r)$ (or $at = (e{:}\psi''(t))^{+} \in \psi(\mathcal{L}r)$). By Lemma A.7, there is an atom $at' \in \mathcal{L}_{\circ}(\varphi(tid, pp(s)))$, such that $at' = e'{:}h$ (respectively $at' = (e'{:}h)^{+}$) and $h \in \{\tilde{m}, \check{m}, \hat{m}\}$, and $at' \succeq at$.

Let $\mathcal{G}_{P} = \langle V, E\rangle$, We have that $\varphi(l), \varphi(l_1) \in P_{\mathcal{P}}$, which implies $\varphi(l), \varphi(l_1) \in V$. The atom at' generates, $\varphi(l) \to \varphi(l)_{e'}, \varphi(l)_{e'} \to h \in E_3$ if $e' \neq \star$. Otherwise, it generates $\varphi(l) \to h \in E_2$. Besides, $h \to \varphi(l_1) \in E_1$ as $at' \succeq at$. In conclusion, we have that $\varphi(l) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_1)$. Any path p of greater length between l and l_1 can be split into smaller paths $p = l \to \cdots \to l_2 \to \cdots \to l_1$ such that $l_2 \in cP_{\mathcal{S}}$. Applying induction hypothesis to the two sub-paths we conclude that $\varphi(l) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_2)$ and $\varphi(l_2) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_1)$ and by transitivity $\varphi(l) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_1)$. □

A.2. Proofs of Section 4

A.2.1. Proof of Theorem 4.16 (soundness of $\mathcal{E}_P$).

In order to prove Theorem 4.16 we will need two results. The firs one expresses that $\mathcal{E}G^r_S$ captures the concurrency information of a given state S:

THEOREM A.9. $\forall S : (S_0 \rightsquigarrow^* S) \Rightarrow (\mathcal{E}^r_S \subseteq \mathcal{E}G^r_S)$

PROOF. Theorem A.9 is equivalent to:

$$\begin{array}{l}\forall S : S_0 \rightsquigarrow^* S : \\ \quad \forall\ tsk(tid_1, bid_1, m_1, l_1, s_1, \mathcal{L}r_1), tsk(tid_2, bid_2, m_2, l_2, s_2, \mathcal{L}r_2) \in S : \\ \quad tid_1 \neq tid_2 : ((tid_1, pp(s_1)), (tid_2, pp(s_2))) \in \mathcal{E}G^r_S\end{array}$$

However, it is sufficient to prove that every task is reachable from the main node ($(0, pp(s_0))$) that corresponds to the main task ($tsk(0, 0, \mathsf{main}, l_0, s_0, \mathcal{L}r)$). This can be expressed:

$$\begin{array}{l}\forall S :, S_i \rightsquigarrow^* S : \\ \exists\ tsk(0, 0, \mathsf{main}, l_0, s_0, \mathcal{L}r) \in S : \\ \forall\ tsk(tid_1, bid_1, m_1, l_1, s_1, \mathcal{L}r_1) \in S : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}_{r_S}}{\rightsquigarrow} (tid_1, pp(s_1))\end{array}$$

In such case, for every two tasks either one of them is the main one and the other is reachable from it or both are different from the main one and they belong to $iMHP$. We can prove the previous property by induction on the states of the program:

Base case:. Straightforward. Only the main task is present. $\forall tsk(tid_1, bid_1, m_1, l_1, s_1, \mathcal{L}r_1) \in Tk : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_1, pp(s_1))$ trivially holds.

Inductive case:. For any possible transition $S \rightsquigarrow S'$. The induction hypothesis is:

$$\exists\, tsk(0, 0, \mathsf{main}, l_0, s_0, \mathcal{L}r) \in S :$$
$$\forall\, tsk(tid_1, bid_1, m_1, l_1, s_1, \mathcal{L}r_1) \in S : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_1, pp(s_1))$$

Although most semantic rules have several effects on the program state, they can be split into steps. Each step is proved to maintain the property. Finally, each semantic rule is expressed as a combination of simple steps.

(1) Sequential step: The new state S' can be obtained through a substitution $S' = S\tau$ of the form:

$$\tau = \{tsk(tid, bid, m, l, s, \mathcal{L}r)/tsk(tid, bid, m, l', s', \mathcal{L}r), obj(bid, f, tid)/obj(bid, f', tid)\}$$

with the condition that both $tsk(tid, bid, m, l, s, \mathcal{L}r)$ and $obj(bid, f, tid)$ belong to S. $\mathcal{G}r_{S'} = \langle V_{S'}, E_{S'}\rangle$ and $\mathcal{G}r_S = \langle V_S, E_S\rangle$ are isomorphic graphs and we can define a graph bijection as a substitution:

$$V'_S = V_S[(tid, pp(s))/(tid, pp(s'))$$

It is easy to see that the given substitution is indeed a bijection. Let $a \rightarrow b$ and edge of $\mathcal{G}r_S$ we have one of the following:

(a) Both a and b are not $(tid, pp(s))$. In this case, $a \rightarrow b$ is in $\mathcal{G}r_{S'}$ as they are not affected by the substitution.

(b) $a = (tid, pp(s))$. This implies that $tsk(tid, bid, m, l, s, \mathcal{L}r) \in TK$ and $\_{:}b \in \mathcal{L}r$ where _ can be a future variable or $\star$. We have that $tsk(tid, bid, m, l', s', \mathcal{L}r) \in S'$ with the same $\mathcal{L}r$ so $(tid, pp(s')) \rightarrow b$ is in $\mathcal{G}r_{S'}$.

(c) $a \rightarrow b = \tilde{tid} \rightarrow (tid, pp(s))$. This implies that $tsk(tid, bid, m, l, s, \mathcal{L}r) \in S$. We have that $tsk(tid, bid, m, l', s', \mathcal{L}r) \in S'$. $\tilde{tid} \rightarrow (tid, pp(s'))$ is in $\mathcal{G}r_{S'}$ by definition.

(d) There cannot be edges of the form $\hat{tid} \rightarrow (tid, pp(s))$ or $\check{tid} \rightarrow (tid, pp(s))$ because they require that $tsk(tid, bid, m, l, s, \mathcal{L}r)$ does not have the lock and that contradicts our condition that $obj(bid, f, tid)$ belong to S.

Once concluded that the graphs are isomorphic the induction hypothesis can be applied to conclude:

$$\exists\, tsk(0, 0, \mathsf{main}, l_0, s_0, \mathcal{L}r) \in S' :$$
$$\forall\, tsk(tid_1, bid_1, m_1, l_1, s_1, \mathcal{L}r_1) \in S' : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_1, pp(s_1))$$

(2) Release:
$S' = S[tsk(tid, bid, m, l, s, \mathcal{L}r)/tsk(tid, bid, m, l, s, \mathcal{L}r'), obj(bid, f, tid)/obj(bid, f, \bot)]$ where $\mathcal{L}r' = \mathcal{L}r[y{:}\check{x}/y{:}\tilde{x}]$. $tsk(tid, bid, m, l, s, \mathcal{L}r)$ and $obj(bid, f, tid)$ have to belong to S initially.
As the $\mathcal{L}r$ sets are always finite, without loss of generality we assume that only one element is substituted. If more than one elements were substituted the same reasoning could be applied repeatedly.
This change has no effect on the graph nodes, $V'_S = V_S$. However, it has an effect on the edges of the graph. By the graph definition we see that changes in a $\mathcal{L}r$ set affect the edges in el_S. $x{:}\check{tid}_1$ is substituted by $x{:}\tilde{tid}_1$:
$el_{S'} = el_S \setminus \{(tid, pp(s)) \rightarrow \check{tid}_1\} \cup \{(tid, pp(s)) \rightarrow \tilde{tid}_1\}$.

Given a task $tsk(tid_2, bid_2, m_2, l_2, s_2, \mathcal{L}r_2) \in S$, by induction hypothesis, there exists $tsk(0, 0, \mathsf{main}, l_0, s_0, \mathcal{L}r) \in S$ such that $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_2, pp(s_2))$. That is, there is a path $p = (0, pp(s_0)) \to x_1 \to x_2 \to \cdots \to (tid_2, pp(s_2))$.
If $(tid, pp(s)) \to \tilde{tid}_1$ does not appear to p, then p is a valid path in $\mathcal{G}r_{_{S'}}$ as every edge in the path belongs to $E_{S'}$ and $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.
If $(tid, pp(s)) \to \tilde{tid}_1$ appears in p. $p = x_1 \to x_2, x_2 \to x_3 \cdots (tid, pp(s)) \to \tilde{tid}_1, \check{tid}_1 \to (tid_1, pp(s_1)) \cdots x_{n-1} \to x_n$. We can create a new path $p' = x_1 \to x_2, x_2 \to x_3 \cdots (tid, pp(s)) \to \tilde{tid}_1, \tilde{tid}_1 \to (tid_1, pp(s_1)) \cdots x_{n-1} \to x_n$.
This new path p' is valid in $\mathcal{G}r_{_{S'}}$: $(tid, pp(s)) \to \tilde{tid}_1$ is the edge added in $el_{S'}$ and $\tilde{tid}_1 \to (tid_1, pp(s_1))$ belongs to both ei_S and $ei_{S'}$ by definition. In conclusion,$(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.
The loss of the lock could make new edges appear in ei_S but that cannot make any path disappear and thus affect the property.

(3) Loss of a future variable association:
$S' = S[tsk(tid, bid, m, l, s, \mathcal{L}r)/tsk(tid, bid, m, lk, l, s, \mathcal{L}r')]$ where $\mathcal{L}r' = \mathcal{L}r[y{:}x/\star{:}x]$. Such substitution does not change the graph as atoms $y{:}x$ and $\star{:}x$ generate the same edges and the nodes remain unchanged.

(4) New task added:

$$S' = S[tsk(tid, bid, m, l, s, \mathcal{L}r)/tsk(tid, bid, m, l, s, \mathcal{L}r')] \cup tsk(tid_1, bid_1, m_1, body(m_1), \emptyset)$$

where $\mathcal{L}r' = \mathcal{L}r \cup \{y{:}\tilde{tid}_1\}$ or $\mathcal{L}r' = \mathcal{L}r \cup \{y{:}\check{tid}_1\}$.
$\mathcal{G}r_{_{S'}} = \langle V', E' \rangle$ where $V' = V \cup \{\tilde{tid}_1, \check{tid}_1, \hat{tid}_1, (tid_1, p_{\hat{m}_1})\}$ and $E' = E \cup \{(tid, s) \to \tilde{tid}_1, \tilde{tid}_1 \to (tid_1, p_{\mathring{m}_1}), \check{tid}_1 \to (tid_1, p_{\mathring{m}_1})$ or $E' = E \cup \{(tid, s) \to \check{tid}_1, \tilde{tid}_1 \to (tid_1, p_{\hat{m}_1}), \check{tid}_1 \to (tid_1, p_{\hat{m}_1})$.
In any case, $\mathcal{G}r_{_{S'}} \supseteq \mathcal{G}r_{_S}$ so any path in $\mathcal{G}r_{_S}$ is still valid in $\mathcal{G}r_{_{S'}}$. Applying induction hypothesis we conclude that for any task $tsk(tid_2, bid_2, m_2, l_2, s_2, \mathcal{L}r_2) \in S$, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.
The only task that is in S' and is not in S is $tsk(tid_1, bid_1, m_1, body(m_1), \emptyset)$. But the program point in this task is reachable from $tsk(tid, bid, m, l, s, \mathcal{L}r')$ as we can create a path p from $(tid, pp(s))$ to $(tid_1, p_{\mathring{m}_1})$: $p = (tid, pp(s)) \to \tilde{tid}_1, \tilde{tid}_1 \to (tid_1, p_{\mathring{m}_1})$ or $p = (tid, pp(s)) \to \check{tid}_1, \check{tid}_1 \to (tid_1, p_{\mathring{m}_1})$ are valid paths depending on the E' that we have.
We have already proved that $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid, pp(s))$ and $(tid, pp(s)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_1, p_{\mathring{m}_1})$. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_1, p_{\mathring{m}_1})$.

(5) Task ending:
$S' = S[tsk(tid, bid, m, l, s, \mathcal{L}r), tsk(tid_1, bid_1, m_1, l_1, \epsilon(v), \mathcal{L}r_1)/tsk(tid, bid, m, l, s, \mathcal{L}r'), tsk(tid_1, bid_1, m_1, l_1, \epsilon(v), \mathcal{L}r_1)]$ where $\mathcal{L}r' = \mathcal{L}r[y{:}\tilde{tid}_1/y{:}\hat{tid}_1\}$.
For a given future variable y there is at most one pair in $\mathcal{L}r$. If there is none, $S' = S$ and the property holds. Otherwise, one pair $y{:}\tilde{tid}_1$ gets substituted by $y{:}\hat{tid}_1$.
This change has no effect on the graph nodes, $V'_S = V_S$. However, it has an effect on the edges of the graph. By the graph definition we see that changes in a $\mathcal{L}r$ set affect the edges in el_S: $el_{S'} = el_S \setminus \{(tid, pp(s)) \to \tilde{tid}_1\} \cup \{(tid, pp(s)) \to \hat{tid}_1\}$
Given a task $tsk(tid_2, bid_2, m_2, l_2, s_2, \mathcal{L}r_2) \in S$, by induction hypothesis $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_2, pp(s_2))$. That is, there is a path p from $(0, s)$ to $(tid_2, pp(s_2))$.

If $p_y \to \tilde{tid}_1$ does not appear to p, then p is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$ and $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, pp(s_2))$.
If $(tid, pp(s)) \to \tilde{tid}_1$ appears p, then $p = x_1 \to x_2, x_2 \to x_3 \cdots (tid, pp(s)) \to \tilde{tid}_1, \tilde{tid}_1 \to (tid_2, p_{\dot{m}_2}) \cdots x_{n-1} \to x_n$. We can create a new path $p' = x_1 \to x_2, x_2 \to x_3 \cdots (tid, pp(s)) \to \hat{tid}_1, \hat{tid}_1 \to (tid_2, p_{\dot{m}_2}) \cdots x_{n-1} \to x_n$.
This new path p' is valid in $\mathcal{G}r_{S'}$ as $(tid, pp(s)) \to \hat{tid}_1$ is the edge added in $el_{S'}$ and $\hat{tid}_1 \to (tid_2, p_{\dot{m}_2})$ belongs to $\mathcal{G}r_{S'}$ and $\mathcal{G}r_S$ by definition. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, pp(s_2))$.

(6) Take lock:
$S' = S[tsk(tid, bid, m, l, s, \mathcal{L}r), obj(bid, f, \bot)/tsk(tid, bid, m, l, s, \mathcal{L}r), obj(bid, f, tid)]$
This transformation can make $\check{tid} \to (tid, p_{\hat{m}}) \in ei_S$ disappear in the case $s = body(m)$ but it will not affect any path between program points due to the lemma A.6. In order to take the lock, it must be free $obj(bid, f, \bot) \in S$. Consequently, $y{:}\check{tid} \notin \mathcal{L}r$ in all the tasks that belong to bid and Node $\check{tid}$ has no incoming edges in $\mathcal{G}r_S$ so there cannot be a path that goes through it.

Finally, we can express the semantic rules as combination of basic steps:

— (NEWOBJECT) is an instance of sequential step (1) with the addition of a new object $obj(bid', f', \bot)$ that does not affect the graph.
— (SELECT) is an instance of Take lock step (6).
— (ASYNC) is an instance of sequential step (1) followed by loss of future variable association (3) and New task added (4).
— (AWAIT1) and (GET) are a sequential step (1) followed by task ending (5).
— (AWAIT2) is a release (2).
— (RELEASE) and (RETURN) are a sequential step (1) followed by a release (2).
— (SEQUENTIAL) is a sequential step (1).

□

The second result states that any pair obtained in the concrete graph of a given state ($\mathcal{E}G_S$) is also obtained by the analysis $\tilde{\mathcal{E}}_P$.

THEOREM A.10. $\forall S : S_0 \rightsquigarrow^* S : \mathcal{E}G_S \subseteq \tilde{\mathcal{E}}_P$

PROOF.
Let $(x', x'_1) \in \mathcal{E}G_S$, there is a $(x, x_1) \in \mathcal{E}G^r_S$ such that $(\varphi(x), \varphi(x_1)) = (x', x'_1)$. By definition of $\mathcal{E}G^r_S$ we have one of the following:

— $(x, x_1) \in dtMHP_S \Leftrightarrow x, x_1 \in cP_S \wedge x \overset{\mathcal{G}r_S}{\rightsquigarrow} x_1$. That means there is a non-empty path $p = xa_1 \cdots a_n x_1$ expressed as a sequence of nodes in $\mathcal{G}r_S$. Using lemma A.8, we conclude that $\varphi(x) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(x_1)$ (which is $x' \overset{\mathcal{G}_P}{\rightsquigarrow} x'_1$) which by the definition of $\tilde{\mathcal{E}}_P$ implies $(x', x'_1) \in \tilde{\mathcal{E}}_P$ □
— $(x, x_1) \in iMHP_S \Leftrightarrow x, y \in cP_S \wedge \exists z \in cP_S(z \rightsquigarrow x \wedge z \rightsquigarrow x_1)$. That is, we have two paths $p_1 = n_1 n_2 \cdots n_s x$ and $p_2 = n'_1 n'_2 \cdots n'_m x$ (where $n'_1 = n_1 = z$) expressed as a sequence of nodes in $\mathcal{G}r_S$. We take the shortest non-common suffix of p_1 and p_2. $p'_1 = n_j n_{j+1} \cdots n_s x$ and $p'_2 = n'_j n'_{j+1} \cdots n'_{m'} x_1$ such that $\forall i (0 < i \leq j : n_i = n'_i) \wedge n_{j+1} \neq n'_{j+1}$. Lets call $z' = n_j = n'_j$. We have that $z' \in cP_S$ as in $\mathcal{G}r_S$ only program point nodes can have more than one outgoing edge. Using lemma A.8, we have that $\varphi(z') \overset{\mathcal{G}_P}{\rightsquigarrow} x'$ and $\varphi(z') \overset{\mathcal{G}_P}{\rightsquigarrow} x'_1$.

We also know that $n_{j+1} \neq n'_{j+1}$, which implies that if $z' = (tid_2, pp(s_2))$, then there exist a task $tsk(tid_2, bid_2, m_2, l_2, s_2, \mathcal{L}r_2) \in S$ such that $at = y{:}a, at' = y'{:}b \in \mathcal{L}r_2, at \neq at'$ are the atoms that generate the edges to n_{j+1} and n'_{j+1}. If $\psi'(at) = \psi'(at')$ we have that $(\psi'(at) = (y{:}\psi'(a))^+ \in \psi(\mathcal{L}r_2)$. Otherwise, $\psi'(at), \psi'(at') \in \psi(\mathcal{L}r_2)$, $\psi'(at) \neq \psi'(at')$. By theorem A.7, either there is an atom $at'' \in \mathcal{L}_\circ(\varphi(tid, pp(s)))$, $at'' = (y''{:}m'')^+$, such that $at'' \succeq \psi'(at)$ and $at'' \succeq \psi'(at')$ or two atoms $at''_1, at''_2 \in \mathcal{L}_\circ(\varphi(tid, pp(s)))$, such that $at''_1 \succeq \psi'(at)$ and $at''_2 \succeq \psi'(at')$. Both cases imply $(x', x'_1) \in \tilde{\mathcal{E}}_P$ □

Finally, Theorems A.9 and A.10 imply the desired soundness of $\mathcal{E}_P$:

PROOF OF THEOREM 4.16 (SOUNDNESS OF $\mathcal{E}_P$).

$$\mathcal{E}_P = \varphi(\mathcal{E}^r_P) = \varphi(\cup_S \mathcal{E}^r_S) \overset{Theorem A.9}{\subseteq} \varphi(\cup_S \mathcal{E}G^r_S) = \cup_S \varphi(\mathcal{E}G^r_S) = \cup_S \mathcal{E}G_S \overset{Theorem A.10}{\subseteq} \tilde{\mathcal{E}}_P$$

□

A.2.2. Proof of Lemma 4.18

PROOF. Let $(pi_1, pi_2) \in p\tilde{\mathcal{E}}_P$ with respect to $iP_{\mathcal{P}}$ we have one of the following:

— $pi_1 \rightsquigarrow pi_2 \in \mathcal{G}_P$. If p is not part of the path, removing p from $\mathcal{G}_P$ does not affect $p\tilde{\mathcal{E}}_P$. Otherwise, the path is: $pi_1 \rightsquigarrow pi_2 =$ "$pi_1 \to x_1 \to \cdots \to x_i \to p \overset{d}{\rightsquigarrow} x_{i+1} \to x_{i+2} \to \cdots \to pi_2$" where $p \overset{d}{\rightsquigarrow} x_{i+1}$ stands for $p \to x_{i+1}$ or $p \to p_y \to x_{i+1}$. We have that $x_i = \tilde{m}$, $x_{i+1} \in \{\tilde{m'}, \check{m'}, \hat{m'}\}$ and $y_1{:}x_{i+1} \in \mathcal{L}_\circ(p)$ (or $(y_1{:}x_{i+1})^+ \in \mathcal{L}_\circ(p)$) where $y_1 \in \{y, \star\}$. If the condition holds there is $p' \in m$ such that $\mathcal{L}_\circ(p) \preceq \mathcal{L}_\circ(p')$ which implies that there is an atom $y'{:}x'_{i+1} \in \mathcal{L}_\circ(p')$ (or $(y'{:}x'_{i+1})^+ \in \mathcal{L}_\circ(p')$) that covers $y_1{:}x_{i+1}$ (respectively $(y_1{:}x_{i+1})^+$). We have that $x_i \to p', p' \overset{d}{\rightsquigarrow} x'_{i+1}, x'_{i+1} \to x_{i+2} \in \mathcal{G}_P$. Consequently, $pi_1 \to x_1 \to \cdots \to x_i \to p' \overset{d}{\rightsquigarrow} x'_{i+1} \to x_{i+2} \to \cdots \to pi_2$ is a valid path in $\mathcal{G}_P$ and p can be removed form $\mathcal{G}_P$.

— $\exists z \in P_{\mathcal{P}} : z \overset{i}{\to} x_1 \rightsquigarrow pi_1 \in \mathcal{G}_P \wedge z \overset{j}{\to} x_2 \rightsquigarrow pi_2 \in \mathcal{G}_P \wedge (x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j = \infty))$. We only have to consider the case where $p = z$.
If $x_1 \neq x_2$ the set $\mathcal{L}_\circ(p)$ must have two atoms $y_1{:}m_1$ and $y_2{:}m_2$ or multiple atoms $(y_1{:}m_1)^+$ and $(y_2{:}m_2)^+$ that generate the edges to x_1 and x_2. If $i = j = \infty$), the set $\mathcal{L}_\circ(p)$ must have one multiple atom $(y_1{:}m_1)^+$ that generates the edge to x_1. If the condition holds there is $p' \in m$ such that $\mathcal{L}_\circ(p) \preceq \mathcal{L}_\circ(p')$ which implies that both atoms (or the multiple atom) are covered by other atoms (or multiple atoms) in $\mathcal{L}_\circ(p')$. Therefore, we have that $p' \overset{i'}{\to} x'_1 \rightsquigarrow pi_1 \in \mathcal{G}_P$, $p' \overset{j'}{\to} x'_2 \rightsquigarrow pi_2 \in \mathcal{G}_P$ and $x'_1 \neq x'_2 \vee (x'_1 = x'_2 \wedge i' = j' = \infty)\}$. That is, if p is a common ancestor of pi_1 and pi_2, then p' is one as well and thus p can be removed from $\mathcal{G}_P$.

□

A.3. Proofs of Section 5

A.3.1. Proof of Theorem 5.10 (Soundness of $\tilde{\mathcal{E}}^H_P$). In order to prove Theorem 5.10 we need two auxiliary results: the first one stating that $\mathcal{L}^H_\circ(p)$ is an approximation of all possible $\mathcal{L}r$ sets of p, and the second one stating that any MHP pair obtained in the concrete graph of a given state ($\mathcal{E}G_S$) is also obtained by the analysis $\tilde{\mathcal{E}}^H_P$.

LEMMA A.11 (SOUNDNESS OF $\mathcal{L}^H_\circ(p)$).

$$\forall S : S_i \rightsquigarrow^* S : tsk(tid, bid, m, l, s, \mathcal{L}r) \in S \Rightarrow \{\psi(\mathcal{L}r)\} \sqsubseteq \mathcal{L}^H_\circ(\varphi(tid, pp(s)))$$

PROOF. Similar to the proof of Th. A.7. □

THEOREM A.12. $\forall S : S_0 \rightsquigarrow^* S : \mathcal{E}G_S \subseteq \tilde{\mathcal{E}}_P^H$

PROOF. The proof is very similar to the one for Th. A.10, by case distinction on the kind of the pair $(x, x_1) \in \mathcal{E}G_S$: $(x, x_1) \in dMHP_S$ or $(x, x_1) \in iMHP_S$. In both cases we will use the soundness of $\mathcal{L}_\circ^H(p)$ (Lemma A.11). □

PROOF OF THEOREM 5.10 (SOUNDNESS OF $\tilde{\mathcal{E}}_P^H$) (SKETCH). We follow a 2-step approach based on the intermediate graphs obtained by the extended semantics in Fig. 15:

$$\mathcal{E}_P = \varphi(\mathcal{E}_P^r) = \varphi(\cup_S \mathcal{E}_S^r) \overset{Th.\ A.9}{\subseteq} \varphi(\cup_S \mathcal{E}G_S^r) = \cup_S \varphi(\mathcal{E}G_S^r) = \cup_S \mathcal{E}G_S \overset{Th.\ A.12}{\subseteq} \tilde{\mathcal{E}}_P^H$$

The first inclusion ($\varphi(\cup_S \mathcal{E}_S^r) \subseteq \varphi(\cup_S \mathcal{E}G_S^r)$) is proven as in the original MHP analysis, using Th. A.9. The pairs in $\cup_S \mathcal{E}G_S$ are a superset of the concrete MHP pairs, but pairs between different branches of **if** statements will not appear. The second inclusion ($\cup_S \mathcal{E}G_S \subseteq \tilde{\mathcal{E}}_P^H$) is proven by Th. A.12. □

A.3.2. Proof of Theorem 5.11. We will need some auxiliary results. The first one states that the conditional transfer function τ_H is more precise than τ:

LEMMA A.13. *If* $H \sqsubseteq \{M\}$ *then* $\tau_H(b, H) \sqsubseteq \{\tau(b, M)\}$

PROOF. Straightforward by definition of τ_H and τ. □

The following lemma states that the result of the conditional method-level analysis $\mathcal{L}_\circ^H(p)$ is more precise than $\mathcal{L}_{\mathcal{P}}$ for every program point.

LEMMA A.14. $\forall p \in P_{\mathcal{P}}.\ \mathcal{L}_\circ^H(p) \sqsubseteq \{\mathcal{L}_{\mathcal{P}}(p)\}$. *It is equivalent to* $\forall p \in P_{\mathcal{P}}.\ \forall M \in \mathcal{L}_\circ^H(p).\ M \sqsubseteq \mathcal{L}_{\mathcal{P}}(p)$.

PROOF SKETCH. Using repeatedly Lemma A.13 from initial statuses $\emptyset \sqsubseteq \{\emptyset\}$. If the instruction is a conditional expressions **if** $cond$ **then** e_1 **else** e_2 we have that:

— $H \sqsubseteq \{M\}$
— $\tau_H(e_1, H) \sqsubseteq \{\tau(e_1, M)\}$
— $\tau_H(e_2, H) \sqsubseteq \{\tau(e_2, M)\}$

By definition of $\sqcup$ we have that $\tau(e_1, M) \sqsubseteq \tau(e_1, M) \sqcup \tau(e_2, M)$ and $\tau(e_2, M) \sqsubseteq \tau(e_1, M) \sqcup \tau(e_2, M)$. Then we have that $\tau_H(e_1, H) \sqcup \tau_H(e_1, H) = \tau_H(e_1, H) \cup \tau_H(e_1, H) \sqsubseteq \{\tau(e_1, M) \sqcup \tau(e_2, M)\}$. If the instruction is a loop expression **while** $cond$ $\{e\}$ then the upper bound operator will merge and obtain an abstract state M that represents any combination of iterations following all the branches, which will be greater than the abstract state H obtained. □

The last result we need is a path preservation lemma stating that connections in the conditional MHP graph $\mathcal{G}_{\mathcal{P}}^H$ are also connections in the MHP graph $\mathcal{G}_{\mathcal{P}}$.

LEMMA A.15 (STEP PRESERVATION).

— *If* $p_1 \to p_n \to m \to p_2 \in \mathcal{G}_{\mathcal{P}}^H$ *then*
  — $p_1 \to m \to p_2 \in \mathcal{G}_{\mathcal{P}}$, *or*
  — $p_1 \overset{\infty}{\to} m \to p_2 \in \mathcal{G}_{\mathcal{P}}$, *or*
  — $p_1 \to p_{1_y} \to m \to p_2 \in \mathcal{G}_{\mathcal{P}}$, *or*
  — $p_1 \to p_{1_y} \overset{\infty}{\to} m \to p_2 \in \mathcal{G}_{\mathcal{P}}$
— *If* $p_1 \to p_n \overset{\infty}{\to} m \to p_2 \in \mathcal{G}_{\mathcal{P}}^H$ *then* $p_1 \overset{\infty}{\to} m \to p_2 \in \mathcal{G}_{\mathcal{P}}$.

PROOF. By the construction of the graphs $\mathcal{G}_{\mathcal{P}}^H$ and $\mathcal{G}_{\mathcal{P}}$, using Lemma A.14. □

LEMMA A.16 (PATH PRESERVATION). *If* $p_1 \rightsquigarrow p_2 \in \mathcal{G}_P^H$ *then* $p_1 \rightsquigarrow p_2 \in \mathcal{G}_P$.

PROOF. By induction on the number of program point nodes in the path, using Lemma A.15. □

Finally, the proof of Theorem 5.11 follows easily from Lemma A.16:

PROOF OF THEOREM 5.11. If $(p_1, p_2) \in directMHP_c$ then $p_1 \rightsquigarrow p_2 \in \mathcal{G}_P^H$ or $p_2 \rightsquigarrow p_1 \in \mathcal{G}_P^H$, so by Lemma A.16 $p_1 \rightsquigarrow p_2 \in \mathcal{G}_P$ (or $p_2 \rightsquigarrow p_1 \in \mathcal{G}_P$) and $(p_1, p_2) \in directMHP \subseteq \tilde{\mathcal{E}}_P$. If $(p_1, p_2) \in indirectMHP_c$ we have two cases:

— $p_3 \rightarrow x_1 \rightarrow x_2 \rightsquigarrow p_1 \in \mathcal{G}_P^H$ and $p_3 \rightarrow x_1 \rightarrow x_3 \rightsquigarrow p_2 \in \mathcal{G}_P^H$ where $x_2 \neq x_3$. In this case by Lemmas A.15 and A.16 we know that there are two paths $p_3 \rightsquigarrow p_1 \in \mathcal{G}_P$ (whose first method node is x_2) and $p_3 \rightsquigarrow p_2 \in \mathcal{G}_P$ (whose first method node is x_3) so $(p_1, p_2) \in indirectMHP \subseteq \tilde{\mathcal{E}}_P$.

— $p_3 \rightarrow x_1 \stackrel{\infty}{\rightarrow} x_2 \rightsquigarrow p_1 \in \mathcal{G}_P^H$ and $p_3 \rightarrow x_1 \stackrel{\infty}{\rightarrow} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P^H$. Therefore by Lemmas A.15 and A.16 we have that $p_3 \stackrel{\infty}{\rightarrow} x_2 \rightsquigarrow p_1 \in \mathcal{G}_P, p_3 \stackrel{\infty}{\rightarrow} x_2 \rightsquigarrow p_2 \in \mathcal{G}_P$, so $(p_1, p_2) \in indirectMHP \subseteq \tilde{\mathcal{E}}_P$.

□