STATIC ANALYSIS OF CONCURRENT OBJECTS

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

Pablo Gordillo Alguacil

TRABAJO FIN DE GRADO EN DOBLE GRADO INGENIERÍA INFORMÁTICA- MATEMÁTICAS. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID
2014-2015

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Junio 2015

Directores:

Elvira Albert
Samir Genaim

Contents

Resumen i

Abstract iv

1 Introduction 1
- 1.1 Contributions . . . 3

2 Preliminaries 5
- 2.1 Language . . . 5
  - 2.1.1 Syntax . . . 5
  - 2.1.2 Operational Semantics . . . 6
- 2.2 Intraprocedural MHP . . . 8
  - 2.2.1 Definition MHP . . . 9
  - 2.2.2 Method-level MHP . . . 10
  - 2.2.3 Application-level MHP . . . 12

3 An Informal Account of our Method 16

4 Must-Have-Finished Analysis 21
- 4.1 Definition of MHF . . . 22
- 4.2 An Analysis to Infer MHF Sets . . . 23

5 MHP Analysis 32
- 5.1 Local MHP . . . 32
- 5.2 Global MHP . . . 35

6 Implementation 40
- 6.1 Implementation Details . . . 41
- 6.2 Computation of MHF Analysis . . . 42

6.3 Computation of LMHP Analysis 43
6.4 MHP Graph 44
6.5 Graphical Interface 44

7 Conclusions, Related and Future Work **48**
7.1 Related Work 48
7.2 Future Work 50

Bibliography **51**

Agradecimientos

Este trabajo supone el final de una etapa. Una etapa llena de buenos momentos rodeado de grandes personas. Han sido cinco años duros, en los que hemos tenido tiempo para todo: divertirnos, agobiarnos, estresarnos y estudiar. Gracias a todos mis compañeros de clase que siempre me han ayudado. No importaba si estábamos en la biblioteca, cafetería o en clase, siempre hemos respondido con una sonrisa haciéndonos los días más llevaderos.

Por otra parte quería agradecer también al grupo COSTA cómo se han portado conmigo, haciéndome sentir desde el principio como uno más. En especial quería dar las gracias Elvira Albert y a Samir Genaim por este año. Siempre han confiado en mi, han estado a mi disposición y me han aguantado sin desesperarse.

Por último no podía olvidarme de mi familia, que siempre están a mi lado, tanto en lo bueno como en lo malo. Se han alegrado por mis éxitos y me han animado en los momentos difíciles. Tengo que mencionar a mi madre y mis hermanos, que son los que han vivido el día a día de estos cinco años, soportando mis tonterías y a mi padre, que en parte soy informático gracias a él.

Gracias a todos.

Resumen

Este trabajo presenta un análisis *may-happen-in-parallel* (puede-ocurrir-en-paralelo) *con sincronización interprocedimental* para lenguajes basados en objetos concurrentes. En este modelo de concurrencia (basado en actores) los objetos son las unidades concurrentes de tal manera que cada objeto es como si tuviese su propio procesador. Un análisis *may-happen-in-parallel* (MHP) infiere los pares de puntos del programa cuyas instrucciones pueden ejecutarse *en paralelo* sobre diferentes componentes distribuidas.

Esta información ha resultado ser esencial para inferir propiedades de completitud (por ejemplo la ausencia de bloqueos) y propiedades de viveza (terminación y consumo de recursos) de programas asíncronos. Los análisis MHP existentes aprovechan los puntos de sincronización para saber que una tarea ha finalizado y que no será ejecutada en paralelo con otras tareas que se encuentren todavía activas.

Nuestro punto de partida es un análisis MHP desarrollado para sincronización *intraprocedimental*, es decir, solo permite la sincronización con tareas que han sido generadas dentro del método actual. El objetivo del presente trabajo es que este análisis MHP soporte sincronización *interprocedimental*, esto es, una tarea generada por otra pueda ser sincronizada (esperada) dentro de otra tarea distinta. Este reto resulta complejo debido a que la sincronización de tareas va más allá de los entornos de los métodos y por tanto las relaciones para inferir esta información requieren de nuevas extensiones para poder capturar las dependencias entre procesos.

En nuestra propuesta se pueden distinguir distintas fases: (1) Una primera donde se lleva a cabo un análisis *must-have-finished* (debe-haber-acabado) mediante el cual se infieren las relaciones de sincronización existentes entre las distintas tareas, (2) una segunda fase local en la que se analiza cada método por separado utilizando la información anteriormente obtenida y (3) una última fase global donde componer todo.

Puesto que el problema es indecidible cuando consideramos un lenguage de programación basado en objetos completo, el análisis calcula una sobre-aproximación del paralelismo real de programas concurrentes. Por último también se ha implementado el análisis incorporándolo en SACO, un analizador de programas concurrentes.

Los principales resultados de este trabajo [5] serán publicados en el Simposio de Análisis Estático 2015: http://sas2015.inria.fr. Se trata del congreso más importante en el área de análisis estático (calificado como A en el ranking de congresos CORE).

Palabras Clave

Must-Have-Finished, May-Happen-in-Parallel, Análisis Estático, Lenguaje ABS, Fórmulas Booleanas, Concurrencia.

Abstract

This work presents a *may-happen-in-parallel* analysis *with inter-procedural synchronization* for languages based in concurrent objects. In this model of concurrency (based on actors) the objects are the concurrency units. The idea behind it is that each object has its own processor. A may-happen-in-parallel (MHP) analysis computes pairs of program points that may execute *in parallel* across different distributed components.

This information has been proven to be essential to infer both safety properties (e.g., deadlock freedom) and liveness properties (termination and resource boundedness) of asynchronous programs. Existing MHP analyses take advantage of the synchronization points to learn that one task has finished and thus will not happen in parallel with other tasks that are still active.

Our starting point is an existing MHP analysis developed for *intra-procedural* synchronization, i.e., it only allows synchronizing with tasks that have been spawned inside the current task. This paper leverages such MHP analysis to handle *inter-procedural* synchronization, i.e., a task spawned by one task can be awaited within a different task. This is challenging because task synchronization goes beyond the boundaries of methods, and thus the inference of MHP relations requires novel extensions to capture inter-procedural dependencies.

We can distinguish different phases in the development of the analysis: (1) The first one where MHF analysis is performed to infer the relations of synchronization that exist between the methods, (2) a second local phase to analyze each method separately and (3) a last phase to composed all information.

As the problem is undecidable when considering a full concurrent objects programming language, the analysis over-approximates the real parallelism programs. Finally, the implementation of the analysis has been integrated in

SACO, a static analyzer of concurrent programs.

The main technical results [5] have been selected to be published in the proceedings of Static Analysis Symposium 2015: http://sas2015.inria.fr. It is the main conference on static analysis (classified as category A in the international ranking CORE).

Key Words

Must-Have-Finished, May-Happen-in-Parallel, Static Analysis, ABS Language, Boolean Formulas, Concurrency.

Chapter 1

Introduction

In order to improve program performance and responsiveness, many modern programming languages and libraries promote an asynchronous programming model, in which *asynchronous* tasks can execute concurrently with their caller tasks, until their callers explicitly wait for their completion. Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables for synchronization [10, 9]. In this model, a method call m on some parameters $\overline{x}$, written as f=m($\overline{x}$), spawns an asynchronous task. Here, f is a *future variable* which allows synchronizing with the termination of the task executing m. The instruction **await** f? allows checking whether m has finished, and blocks the execution of the current task if m is still running. As concurrently-executing tasks interleave their accesses to shared memory, asynchronous programs are prone to concurrency-related errors [7]. Automatically proving safety and liveness properties still remains a challenging endeavor today. In this model task scheduling is cooperative (or non-preemptive), i.e, switching between task of the same object happens only at specific scheduling points during the execution.

MHP is an analysis of utmost importance to ensure both liveness and safety properties of concurrent programs. The analysis computes *MHP pairs*, which are pairs of program points whose execution might happen in parallel across different distributed components. In this fragment of code f=m(..) ;...; **await** f?; the execution of the instructions of the asynchronous task m may happen in parallel with the instructions between the asynchronous call and the **await**. However, due to the **await** instruction, the MHP analysis is able to ensure that they will not run in parallel with the instructions after the **await**. This piece of information is fundamental to prove more complex

properties: in [11], MHP pairs are used to discard unfeasible deadlock cycles. In order to see it, consider the methods f and k that contain as unique instruction :

$$\mathsf{f()\{\ z{=}1;\ \}} \quad \mathsf{k()\{\ z{=}3;\ \}}.$$

Method g receives a future variable as its argument and uses it for synchronization inside its body:

$$\mathsf{g(w)\{await\ w?;\}}.$$

Finally there is a main method m that contains the sequence of instructions:

$$\mathsf{m()\{x\ =\ o1!f();\ y\ =\ o2!g(x);\ await\ y?;\ v\ =\ o1!k();\}}.$$

What is relevant in the example is the synchronization of the tasks. If we can ensure that f and k will not run in parallel the program is deadlock free. We are able to prove this when our MHP analysis with inter-procedural synchronization is used. The MHP analysis with intra-procedural synchronization will inaccurately infer that f and k can run in parallel and a deadlock will be detected as the object o1 is supposed to be blocked executing the method f.

As another application, in [4], the use of MHP pairs allows proving termination and inferring the resource consumption of loops with concurrent interleavings. As simple examples, first consider a procedure g that contains as unique instruction:

$$\mathsf{g()\{y{=}{-}1\}},$$

where y is a global variable. The following loop:

$$\mathsf{y{=}1;\ while(i{>}0)\{i{=}i{-}y;\}}$$

might not terminate if g runs in parallel with it, since g can modify y to a negative value and the loop counter will keep on increasing. However, if we can guarantee that g will not run in parallel with this code, we can ensure termination and resource-boundedness for the loop.

1.1 Contributions

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

The main contributions of this thesis are:

- The enhancement to inter-procedural synchronization which requires the following relevant extensions to the analysis:

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

- We have implemented our approach in SACO [2], a static analyzer for ABS programs which is able to infer the aforementioned liveness

and safety properties. ABS [12] is an asynchronous language that has been proposed to model distributed concurrent objects and allows inter-procedural and intra-procedural synchronization. SACO has a web interface and can be used online at http://costa.ls.fi.upm.es/saco/web, where the examples used in the thesis are also available.

Chapter 2

Preliminaries

2.1 Language

Our analysis is formalized for an abstract model that includes procedures, asynchronous calls, and future variables [10, 9]. It also includes conditional and loop constructs, however, conditions in these constructs are simply non-deterministic choices. Developing the analysis at such abstract level is convenient [13], since the actual computations are simply ignored in the analysis and what is actually tracked is the control flow that originates from asynchronously calling methods and synchronizing with their termination. Our implementation, however, is done for the full concurrent object-oriented language ABS [12] (see Chapter 6).

2.1.1 Syntax

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

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

Here all variables are future variables, which are used to synchronize with the termination of the called methods. Those future variables that are used in a method but are not in its parameters are the *local future variables* of the method (thus we do not need any special instruction for declaring them). In loops and conditions, the symbol $*$ stands for non-deterministic choice (*true* or *false*). The instruction $y = m(\bar{x})$ creates a new task which executes method m, and binds the future variable y with this new task so we can

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

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

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

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

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

Figure 2.1: Derivation Rules

synchronize with its termination later. Inter-procedural synchronization is realized in the language by passing future variables as parameters, since the method that receives the future variable can await for the termination of the associated task (created outside its scope). For simplifying the presentation, we assume that *method parameters are not modified inside each method.* For a method m, we let P_m be the set of its parameters, L_m the set of its local variables, and $V_m = P_m \cup L_m$.

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

2.1.2 Operational Semantics

Next we define a formal (interleaving) operational semantics for our language. A task is of the form $tsk(tid, l, s)$ where tid is a unique identifier, l is a mapping from local variables and parameters to task identifiers, and s is a sequence of instructions. Local futures are initialized to $\perp$. A state S is a set

of tasks that are executing in parallel. From a state S we can reach a state S' in one execution step, denoted $S \rightsquigarrow S'$, if S can be rewritten using one of the derivation rules of Figure 2.1 as follows: if the conclusion of the rule is $A \rightsquigarrow B$ such that $A \subseteq S$ and the premise holds, then $S' = (S \backslash A) \cup B$. The meaning of the derivation rules is quite straightforward: (SKIP) advances the execution of the corresponding task to the next instruction; (IF) nondeterministically chooses between one of the branches; (LOOP) nondeterministically chooses between executing the loop body or advancing to the instruction after the loop; (CALL) creates a new task with a fresh identifier tid', initializes the formal parameters $\bar{z}$ of m to those of the actual parameters $\bar{x}$, sets future variable y in the calling task to tid', so one can synchronize with its termination later (other local futures are assumed to be $\perp$); and (AWAIT) advances to the next instruction if the task associated to x has terminated already. Note that when a task terminates, it does not disappear from the state but rather its sequence of instructions remains empty.

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

Example 2.1. *Figure 2.2 shows some simple examples in our language. Methods* m_1, m_2 *and* m_3 *are* main *methods and the remaining ones are auxiliary. Let us consider some steps of one possible derivation from* m_2*:*

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

In S_1 *we execute until the asynchronous call to* f *which creates a new task identified as* 1 *and binds* x *to this new task. In* S_2 *we have executed the* **skip** *and the asynchronous invocation to* g *that adds in the new task the binding of the formal parameter* w *to the task identified as* 1*. In* S_3 *we proceed with the execution of the instructions in* m_2 *until reaching the* **await** *that blocks this task until* g *terminates. Also, in* S_3 *we have executed entirely* f *(denoted by* ϵ*).* S_4 *proceeds with the execution of* g *whose* **await** *can be executed since task* 1 *is at its exit point* ϵ*.*

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

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

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

```
37 f() {
38   skip;
39 }
40
41 q() {
42   skip;
43 }
44
45 g(w) {
46   skip;
47   await w?
48   skip;
49 }
50
```

```
51 h(a,b) {
52   skip;
53   z=g(a);
54   skip;
55   await z?;
56 }
57
58 k(a,b) {
59   skip;
60   await a?;
61   skip;
62   await b?;
63   skip;
64 }
```

Figure 2.2: Examples for MHP analysis (m_1, m_2, m_3 are main methods).

2.2 Intraprocedural MHP

In this section we summarize the intraprocedural MHP analysis of Albert et al. [3], which is the basis to our work. We first formally define the property MHP since we aim at approximating it later as well. After that, the two main steps in which the analysis is done are presented: the method-level MHP and the application-level MHP.

2.2.1 Definition MHP

We assume that instructions are labeled such that it is possible to obtain the corresponding program points identifiers. $p_{\hat{m}}$ is used to refer to the entry program point of method m, which is typically that of its first instruction, and ℓ_m to refer to an exit program point which is reached after executing the last instruction of m. The set of all program points of P is denoted by $\texttt{ppoints}(P)$. $p \in m$ indicates that program point p belongs to method m.

In what follows, given a task $tsk(tid, l, s)$, we let $pp(s)$ be the program point of the first instruction in s. When s is an empty sequence, $pp(s)$ refers to the exit program point of the corresponding method.

Definition 2.1 (MHP pairs)**.** *Given a state S, we define its set of MHP pairs, i.e., the set of program points that execute in parallel in S, as $\mathcal{E}(S) = \{(pp(s_1), pp(s_2)) \mid tsk(tid_1, l_1, s_1), tsk(tid_2, l_2, s_2) \in S, tid_1 \neq tid_2\}$.*

The set of MHP pairs for a program P is then defined as the the set of MHP pairs of all reachable states, namely $\mathcal{E}_P = \cup\{\mathcal{E}(S_n) \mid S_0 \rightsquigarrow^* S_n\}$.

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.

Example 2.2. *We have the following MHP pairs in the fragment of the derivation shown in Example 2.1, among many others: from S_1 we have (16,38) that captures that the first instruction of* f *executes in parallel with the instruction 16 of method* m2 *in Figure 2.2, from S_2 we have (18,38) and (18,46). The important point is that we have no pair (20,38) since when the* **await** *at line 19 executes at S_4, it is guaranteed that* f *has finished. This is due to the inter-procedural dependency at program point 47 of* g *where the task* f *is awaited: variable* x *is passed as argument to* g*, which allows* g *to synchronize with the termination of* f *at line 47 even if* f *was called in a different method.*

Let us explain now 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 2.2. We consider an intra-procedural MHP analysis. There is a method m_2 which calls methods f, g. We consider a call to m_2 and no other process executing:

- In the example, both f and g are called from m_2. The **await** instruction at program point 19 ensures that g will have finished afterwards. However, the call to f from m_2 might be still executing. We say that f might *escape* from m_2. Method calls that might escape need to be considered.

- Both f and g are called from m_2. Any program point of f might execute in parallel with g even if they do not call each other, that is, they have an *indirect* MHP relation. Furthermore, program point 38 might execute in parallel with any program point of m_2. We say that m_2 is a common *ancestor* of f and g. Two methods execute indirectly in parallel if they have a common ancestor.

2.2.2 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).

The underlying abstract states that are used in the analysis are multisets of symbolic values, that describe the status of all tasks invoked so far. 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, it is added to the multiset, 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? instruction is executed, the status of the corresponding method is changed to finished. Next this intuition is applied to the methods of Figure 2.2.

Example 2.3. *Consider methods in Figure 2.2. In* m_2*, the call to* f *at line 15 creates a task that is* active *up to the end of the method. The call to* g *at line 17 creates an* active *task that becomes* finished *at line 19.*

In the rest of this section the method-level analysis is formalized.

Definition 2.2 (MHP atoms)**.** *An MHP atom is a symbolic expression of the form* $y{:}\tilde{m}$*,* $y{:}\hat{m}$ *or* $y{:}\check{m}$*, where* m *is a method name and* y *is either a future variable or* $\star$*. The set of all MHP atoms is defined as* $\mathcal{A} = \{y{:}x \mid m \in \texttt{methods}(P), x \in \{\tilde{m}, \hat{m}, \check{m}\}, y \in \texttt{futures}(P) \cup \{\star\}\}$*.*

The symbolic values in the above definition are used to describe the status of a task as follows:

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; and
3. $y{:}\check{m}$ describes a *pending* task that is an instance of method m.

In the three cases above, the 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.

Definition 2.3 (asbtract MHP states)**.** *An abstract MHP state* M *is a multiset of MHP atoms from* $\mathcal{A}$*. The set of all multisets over* $\mathcal{A}$ *is denoted by* $\mathcal{B}$*.*

In addition, several tasks can be associated to the same future variable meaning that at most one of them can be available at the same time (since only one task can be associated to a future variable in the semantics). The use of multisets allows modelling the information that several instances of the same method might be running in parallel, which is particularly useful when methods are called within loops. $(a, i) \in M$ indicates that a appears exactly $i > 0$ times in M. i is omitted when it is equal to 1 and when testing for inclusion. All operations on sets, e.g., union and intersection, refer to operations on multisets unless explicitly stated otherwise.

$$
\begin{array}{lrll}
(1) & \tau(y{=}m(\bar{x}), M) = & M[y{:}z/\star{:}z] \cup \{y{:}\tilde{m}\} & z \in \{\check{p}, \tilde{p}, \hat{p}\} \\
(2) & \tau(\textsf{await } y?, M) = & M[y{:}z/y{:}\hat{m}] & z \in \{\check{m}, \tilde{m}\} \\
(3) & \tau(b, M) = & M & \text{otherwise}
\end{array}
$$

Figure 2.3: Method-level MHP transfer function: $\tau : s \times \mathcal{B} \mapsto \mathcal{B}$.

The effect of executing each instruction on a given abstract state is given by means of a transfer function that is depicted in Figure 2.3.

The analysis of a program P is done as follows. For each method $m \in \texttt{methods}(P)$ it starts from an abstract state $\emptyset \in \mathcal{B}$, which assumes that there are no tasks executing, and propagates the information to the different program points by applying the transfer function τ defined in Figure 2.3 on the code $body(m)$.

Example 2.4. *The following information summarizes the result of the analysis for some selected program points of interest of method* m_2 *from Figure 2.2:*

$$\begin{array}{rr} L13 : \{\} & L20 : \{x{:}\tilde{f}, z{:}\hat{g}\} \\ L15 : \{\} & L21 : \{x{:}\tilde{f}, z{:}\hat{g}\} \\ L16 : \{x{:}\tilde{f}\} & L39 : \{\} \\ L18 : \{x{:}\tilde{f}, z{:}\tilde{g}\} & L48 : \{\} \end{array}$$

The state associated to a program point represents the state before the execution of the corresponding instruction. The result for the entry point 13 is $\emptyset$*. When we reach program point 16, there is a call to* `f` *and applying the rule (1) from Figure 2.3, the abstract state changes to include the MHP atom* $x{:}\tilde{f}$*. At program point 20 the rule (2) from Figure 2.3 is applied and changes the state of* `g` *to finished. Program point 21 is the exit point of method* m_2 *and the abstract state associated to it shows that the execution of method* `g` *has finished but* `f` *is still executing and therefore, it is a escaped method.*

2.2.3 Application-level MHP

Next the notion of *MHP graph* is introduced. 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 program point nodes to method nodes represent tasks that any 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 outedges of program point nodes. When two nodes are directly connected by $i > 0$ edges, they are connected with a single edge of weight i, and thus it is convenient to consider MHP graphs as weighted graphs.

Definition 2.4 (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:*

$$\begin{aligned} V = \; & \{\tilde{m}, \hat{m}, \check{m} \mid m \in \texttt{methods}(P)\} \cup \texttt{ppoints}(P) \cup \\ & \cup \{p_y \mid p \in \texttt{ppoints}(P), y{:}m \in \mathcal{L}_{P}(p)\} \end{aligned}$$

$$\begin{aligned} E_1 = \; & \{\tilde{m} \xrightarrow{0} p \mid m \in \texttt{methods}(P), p \in m\} \cup \\ & \cup \{\hat{m} \xrightarrow{0} \ell_m, \check{m} \xrightarrow{0} p_{\hat{m}} \mid m \in \texttt{methods}(P)\} \end{aligned}$$

$$E_2 = \; \{p \xrightarrow{i} x \mid p \in \texttt{ppoints}(P), (\star{:}x, i) \in \mathcal{L}_{P}(p)\}$$

$$E_3 = \; \{p \xrightarrow{0} p_y, p_y \xrightarrow{i} x \mid p \in \texttt{ppoints}(P), (y{:}x, i) \in \mathcal{L}_{P}(p)\}$$

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

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

1. Edges in E_1 describe the program points at which each task can be, depending on its status. Each m contributes the edges (a) $\tilde{m} \xrightarrow{0} p$ for each $p \in m$, which means that if m is active it can be at any program point – but only at one; (b) $\check{m} \xrightarrow{0} p_{\hat{m}}$, which means that when m is pending, it is at the entry program point; and (c) $\hat{m} \xrightarrow{0} \ell_m$, which means that when m is finished, it is at the exit program point;

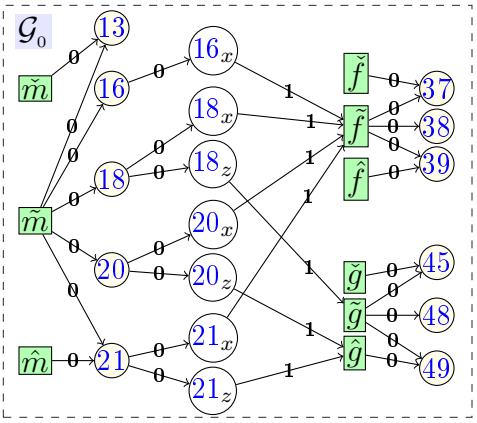

Figure 2.4: MHP graph of method m_2 in Figure 2.2

2. Edges in E_2 describe which tasks might run in parallel at each program point. For every program point $p \in \texttt{ppoints}(P)$, if $(\star{:}x, i) \in \mathcal{L}_P(p)$ then $p \xrightarrow{i} x$ is added to E_2. Such edge means, if $x = \tilde{m}$ for example, that up to i instances of m might be running in parallel when reaching p. Note that i is the multiplicity of the edge, i.e., we could copy the edge i times instead;

3. Edges in E_3 enrich the information for each program point given in E_2. An edge $p_y \xrightarrow{i} x$ is added to E_3 if $(y{:}x, i) \in \mathcal{L}_P(p)$. For each future variable y that appears in $\mathcal{L}_P(p)$ an edge $p \xrightarrow{0} p_y$ is also added to E_3. This allows us to accurately handle cases in which several MHP atoms in $\mathcal{L}_P(p)$ are associated to the same future variable. Note that in this case the weight i will always be 1 (assuming redundant elements are removed).

Note that MHP graphs might have cycles due to recursion.

Inference of Global MHP

Given the MHP graph $\mathcal{G}_P$, two program points $p_1, p_2 \in \texttt{ppoints}(P)$ 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 $p_3 \in \texttt{ppoints}(P)$, and non-empty paths from p_3 to p_1 and from p_3 to p_2 that are either different in the first edge, or they share the first edge but it has weight $i > 1$.

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task that was invoked within the task executing p_1 and from which it is possible to *transitively* reach p_2, or vice-versa. 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.

$p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\scriptscriptstyle P}$ indicates that there is a path of length at least 1 from p_1 to p_2 in $\mathcal{G}_{\scriptscriptstyle P}$, and $p_1 \xrightarrow{i} x \rightsquigarrow p_2$ indicates that such path starts with an edge to x with weight i.

Definition 2.5. *The MHP information induced by the MHP graph $\mathcal{G}_{\scriptscriptstyle P}$ is defined as $\tilde{\mathcal{E}}_P = directMHP \cup indirectMHP$ where*

$$\begin{aligned} directMHP = \quad & \{(p_1, p_2) \mid \quad p_1, p_2 \in \texttt{ppoints}(P), p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\scriptscriptstyle P})\} \\ indirectMHP = \quad & \{(p_1, p_2) \mid \quad p_1, p_2, p_3 \in \texttt{ppoints}(P), p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_{\scriptscriptstyle P}, \\ & \qquad\qquad p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_{\scriptscriptstyle P}, x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\} \end{aligned}$$

Example 2.5. *In Figure 2.4 we show the graph for the main m_2 of Figure 2.2. It has been built using the information of Example 2.4. For each method, there are three nodes to represent its possible states (active, pending or finished). There are also future variable nodes that represent the local future variables of method m_2. Only program points of interest have been included in the graph. The edges that connect the future variable nodes with method nodes are labeled with 1 as it is the multiplicity of all MHP atoms. Program point 39 can run in parallel with program point 21 and the direct path that connects both nodes represents that* `f` *is a escaped method. Program point 39 can run in parallel with program point 48 as the nodes that represent these program points have as ancestor the node represented by program point 18.*

Chapter 3

An Informal Account of our Method

In this chapter, we provide an overview of our method by explaining the analysis of m_2 from Figure 2.2. Our goal is to infer precise MHP information that describes, among others, the following representative cases:

(1) any program point of g cannot run in parallel with program point 20, because at program point 19 method m_2 awaits for g to terminate;

(2) program point 38 cannot run in parallel with program point 20, since when waiting for the termination of g at program point 19 we know that f *must-have-finished* as well due to the *dependency* relation that arises when m_2 implicitly waits for the termination of f; and

(3) program point 38 cannot run in parallel with program point 48, because f *must-have-finished* due to the synchronization on the local future variable w at program point 47 that refers to future variable x of m_2.

Let us first informally explain which MHP information the analysis of Section 2.2 is able to infer for m_2, and identify the reasons why it fails to infer some of the desired information. As we have seen, the analysis of Section 2.2 is carried out in two phases: (1) each method is *analyzed separately* to infer local MHP information; and (2) the local information is used to construct a global MHP graph from which MHP pairs are extracted by checking reachability conditions among the nodes.

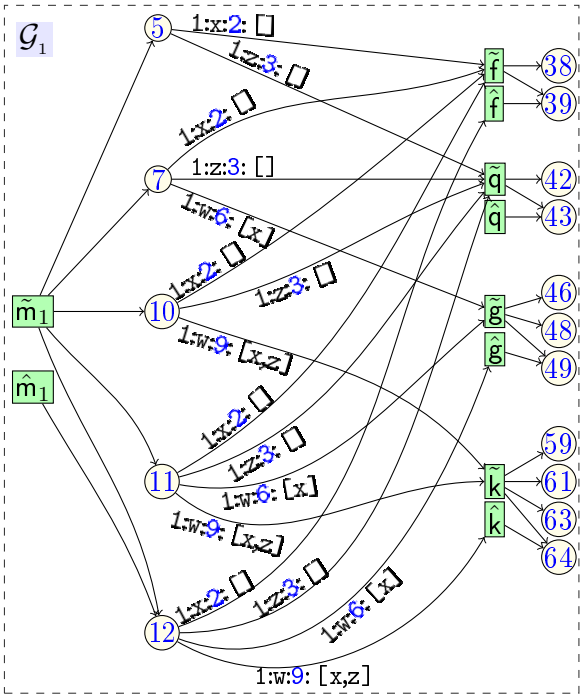

Figure 3.1: MHP graph $\mathcal{G}_1$ corresponds to analyzing m_1 from Figure 2.2

The local analysis infers, for each program point, a multiset of MHP atoms where each atom describes a task that might be executing in parallel when reaching that program point, but only considering tasks that have been invoked locally within the analyzed method. As we have already said in Section 2.2, an atom of the form $x{:}\tilde{m}$ indicates that there might be an *active* instance of m executing at any of its program points, and is bound to the future variable x. An atom of the form $x{:}\hat{m}$ differs from the previous one in that m must be at its exit program point, i.e., has finished executing already. For method m_2, the local MHP analysis infers, among others, $\{\mathsf{x}{:}\tilde{\mathsf{f}}\}$ for program point 16, $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\tilde{\mathsf{g}}\}$ for program point 18, and $\{\mathsf{x}{:}\tilde{\mathsf{f}}, \mathsf{z}{:}\hat{\mathsf{g}}\}$ for program point 20 and program point 21, because g has been awaited locally. Observe that the sets of program point 20 and program point 21 include $\mathsf{x}{:}\tilde{\mathsf{f}}$ and not $\mathsf{x}{:}\hat{\mathsf{f}}$, although method f has finished already when reaching program point 20 and program point 21 (since g has finished). This information cannot be inferred by the local analysis of Section 2.2 since it is applied to each method

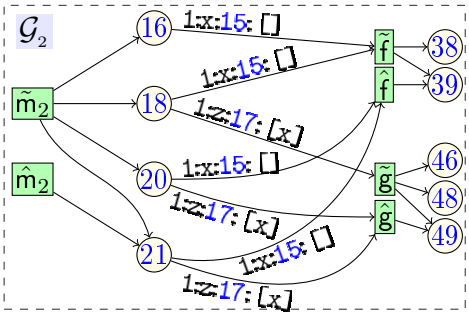

Figure 3.2: MHP graph $\mathcal{G}_2$ corresponds to analyzing m_2.

separately, ignoring (a) transitive (non-local) calls and (b) inter-procedural synchronizations.

In the second phase, the analysis of Section 2.2 builds the MHP graph whose purpose is to capture MHP relations due to transitive calls (point (a) above). The graph $\mathcal{G}_0$ depicted in Figure 2.4 for m_2 is constructed as it has been explained in Section 2.2.3. For simplicity we include only program points of interest.

The MHP pairs are obtained from $\mathcal{G}_0$ (Figure 2.4), using the principle from Definition 2.5. Applying this principle to $\mathcal{G}_0$, we can conclude that program point 20 cannot execute in parallel with any program point of g, which is precise as expected, and that program point 20 can execute in parallel with program point 38 which is imprecise. This imprecision is attributed to the fact that the MHP analysis of Section 2.2 does not track inter-method synchronizations.

To overcome the imprecision, we develop a must-have-finished analysis that captures inter-method synchronizations, and use it to improve the two phases of Section 2.2. This analysis will infer, for example, that "*when reaching* program point 48*, it is guaranteed that whatever task bound to* w *has finished already*", and that "*when reaching* program point 20*, it is guaranteed that whatever tasks bound to* x *and* z *have finished already*". By having this information at hand, the first phase of Section 2.2 can be improved as follows: when analyzing the effect of **await** z? at program point 20, we change the status of both g and f to finished, because we know that any task bound z and x has finished already. This will require to enrich the information of the MHP atoms as follows: an MHP atom will be of the form $y{:}\ell{:}\tilde{m}(\bar{x})$ or

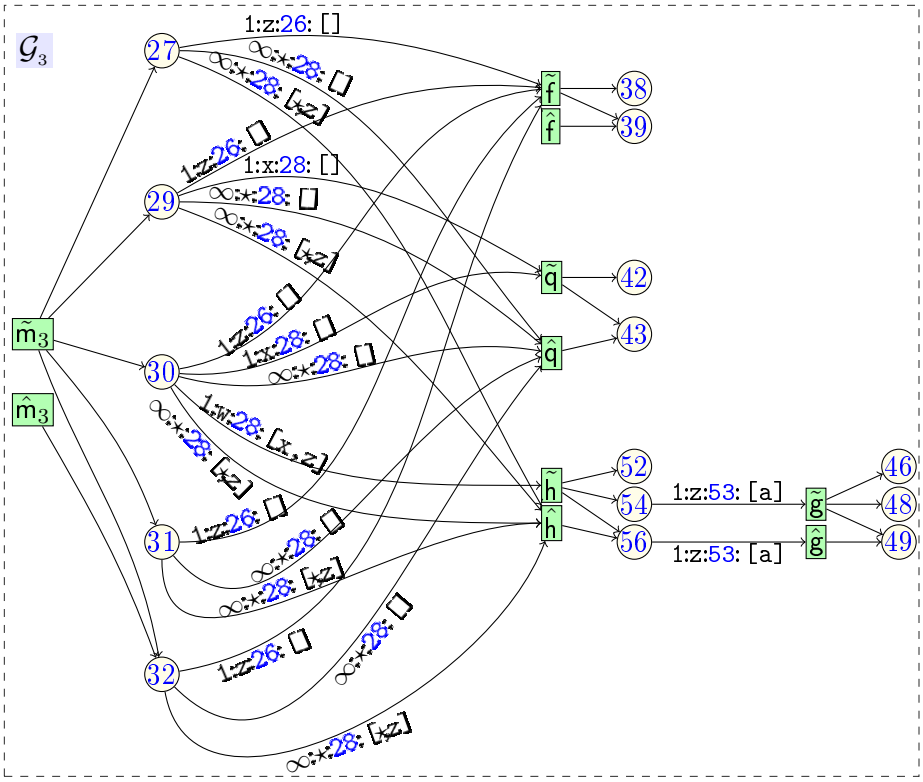

Figure 3.3: MHP graph $\mathcal{G}_3$ corresponds to analyzing m_3 from Figure 2.2.

$y{:}\ell{:}\hat{m}(\bar{x})$, where the new information ℓ and $\bar{x}$ are the calling site and the parameters passed to m. In summary, the modified first phase will infer $\{\mathsf{x}{:}15{:}\tilde{\mathsf{f}}()\}$ for L16, $\{\mathsf{x}{:}15{:}\tilde{\mathsf{f}}(), \mathsf{z}{:}17{:}\tilde{\mathsf{g}}(\mathsf{x})\}$ for L18, and $\{\mathsf{x}{:}15{:}\hat{\mathsf{f}}(), \mathsf{z}{:}17{:}\hat{\mathsf{g}}(\mathsf{x})\}$ for program points 20 and 21.

In the second phase of the analysis:

(i) the construction of the MHP graph is modified to use the new local MHP information; and

(ii) the principle used to extract MHP pairs is modified to make use to the must-have-finished information.

The new MHP graph constructed for m_2 is depicted in Figure 3.2 as $\mathcal{G}_2$. Observe that the labels on the edges include the new information available in the MHP atoms. Note that when two paths are labeled with the same future variable, it is because there is a disjunction (e.g., from an if-then-else) and only one of the paths might actually occur. Importantly, the spurious MHP information that is inferred by Section 2.2 is not included in this graph:

(1) in contrast to $\mathcal{G}_0$, $\mathcal{G}_2$ does not include paths from node 20 and 21 to $\tilde{\mathsf{f}}$, but to $\hat{\mathsf{f}}$. This implies that program point 38 cannot run in parallel with program point 20 or program point 21;

(2) in $\mathcal{G}_2$, we still have paths from 18 to 38 and 48, which means, if the old principle for extracting MHP pairs is used, then program point 38 and program point 48 might happen in parallel. The main point is that, using the labels on the edges, we know that the first path uses a call to f that is bound to x, and that this same x is passed to g, using the parameter w, in the first edge of the second path. Now since the must-have-finished analysis tell us that at program point 48 any task bound w is finished already, we conclude that f must be at its exit program point and thus this MHP pair is spurious (because program point 38 is not an exit program point).

Chapter 4

Must-Have-Finished Analysis

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

Example 4.1. *The MHF sets for the program points of Figure 2.2 are:*

$L2$: {x,w,z}	$L10$: {}	$L18$: {}	$L29$: {w}	$L43$: {}	$L54$: {}	$L63$: {a,b}
$L3$: {z,w}	$L11$: {}	$L19$: {}	$L30$: {}	$L46$: {}	$L55$: {}	$L64$: {a,b}
$L4$: {w}	$L12$: {x,w}	$L20$: {x,z}	$L31$: {x,w}	$L47$: {}	$L56$: {a,z}	
$L5$: {w}	$L14$: {x,z}	$L21$: {x,z}	$L32$: {x,w}	$L48$: {w}	$L59$: {}	
$L6$: {w}	$L15$: {x,z}	$L26$: {x,z,w}	$L38$: {}	$L49$: {w}	$L60$: {}	
$L7$: {}	$L16$: {z}	$L27$: {x,w}	$L39$: {}	$L52$: {z}	$L61$: {a}	
$L9$: {w}	$L17$: {z}	$L28$: {x,w}	$L42$: {}	$L53$: {z}	$L62$: {a}	

At program points that correspond to method entries, all local variables (but not the parameters) are finished since they point to no task. For g*: at program point 46 and program point 47 no task is guaranteed to have finished, because the task bound to* w *might be still executing; at program point 48 and program point 49, since we passed through* **await** w*? already, it is guaranteed that* w *is finished. For* k*: at program point 59 and program point 60 no task is guaranteed to have finished; at program point 61 and program point 62* a *is finished since we already passed through* **await** a*?; and at program point 63 and program point 64 both* a *and* b *are finished. For* m_1*: at program point 12 both* w *and* x *are finished. Note that* w *is finished due to* **await** w*?, and* x *is finished due to the implicit dependency between the termination of* x *and* w*.*

4.1 Definition of MHF

By carefully examining the MHF sets of Ex. 4.1, we can see that an analysis that simply tracks MHF sets would be imprecise. For example, since the MHF set at program point 11 is empty, the only information we can deduce for program point 12 is that w is finished. To deduce that x is finished we must track the implicit dependency between w and x. Next we define a more general MHF property that captures such dependencies, and from which we can easily compute the MHF sets.

Definition 4.1. *Given a program point* $\ell \in \texttt{ppoints}(P)$*, we let* $\mathcal{F}(\ell) = \{f(S_i, l) \mid S_0 \rightsquigarrow^* S_i, tsk(tid, l, s) \in S_i, pp(s) = \ell\}$ *where* $f(S, l) = \{x \mid x \in dom(l),\ l(x) = \bot \vee (l(x) = tid' \wedge tsk(tid', l', \epsilon) \in S)\}$.

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

Example 4.2. *The values of* $\mathcal{F}(\ell)$ *for all program points from Figure 2.2 are:*

L2 : {{w,x,z}}
L3 : {{w,x,z},{w,z}}
L4 : {{w,x,z},{w,z},{w,x},{w}}
L5 : {{w,x,z},{w,z},{w,x},{w}}
L6 : {{w,x,z},{w,z},{w,x},{w}}
L7 : {{w,x,z},{w,x},{x,z},{z},{x},{}}
L9 : {{w,x,z},{w,z},{w,x},{w}}
L10: {{w,x,z},{x,z},{z},{x},{}}
L11: {{w,x,z},{w,x},{x,z},{z},{x},{}}
L12: {{w,x,z},{w,x}}
L14: {{x,z}}
L15: {{x,z}}
L16: {{x,z},{z}}
L17: {{x,z},{z}}
L18: {{x,z},{x},{}}
L19: {{x,z},{x},{}}
L20: {{x,z}}
L21: {{x,z}}
L26: {{w,x,z}}
L27: {{w,x,z},{w,x}}
L28: {{w,x,z},{w,x}}
L29: {{w,x,z},{w,x},{w,z},{w}}

$L30$: {{w,x,z},{w,x},{x,z},{x},{z},{}}
$L31$: {{w,x,z},{w,x}}
$L32$: {{w,x,z},{w,x}}
$L38$: {{}}
$L39$: {{}}
$L42$: {{}}
$L43$: {{}}
$L46$: {{w},{}}
$L47$: {{w},{}}
$L48$: {{w}}
$L49$: {{w}}

$L52$: {{a,b,z},{a,z},{b,z},{z}}
$L53$: {{a,b,z},{a,z},{b,z},{z}}
$L54$: {{a,b,z},{a,z},{b,z},{a,b},{a},{b},{}}
$L55$: {{a,b,z},{a,z},{b,z},{a,b},{a},{b},{}}
$L56$: {{a,b,z},{a,z}}
$L59$: {{a},{b},{a,b}}
$L60$: {{a},{b},{a,b}}
$L61$: {{a},{a,b}}
$L62$: {{a},{a,b}}
$L63$: {{a,b}}
$L64$: {{a,b}}

For program point 5 different sets arise by considering all possible orderings in the execution of tasks f, q *and* m_1, *but* $\texttt{mhf}(L5) = \{w\}$ *which means that* w *is always finished when reaching program point 5. Note that for any* $F \in \mathcal{F}(L11)$, *if* $w \in F$ *then* $x \in F$, *which means that if* w *is finished at program point 11, then* x *must have finished as well.*

4.2 An Analysis to Infer MHF Sets

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

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

to *false* is a satisfying assignment for Φ. The set of all models of Φ is denoted $[\![\Phi]\!]$. The set of all MHF states for m, together with the formulas *true* and *false*, is denoted $\mathcal{A}_m$.

Example 4.3. *Assume $V_m = \{x, y, z\}$. The Boolean formula $x \vee y$ states that either x or y or both are finished, and that z can be in any status. This information is precisely captured by the models $[\![x \vee y]\!] = \{\{x\},\{y\},\{x,y\},\{x,z\},\{y,z\},\{x,y,z\}\}$. The Boolean formula $z \wedge (x \to y)$ states that z is finished, and if x is finished then y is finished. This is reflected in $[\![z \wedge (x \to y)]\!] = \{\{z\}, \{z, y\}, \{z, x, y\}\}$ since z belongs to all models, and any model that includes x includes y as well. The formula false means that the corresponding program point is not reachable. The following MHF states correspond to all program points from Figure 2.2:*

Φ_2 : $w \wedge x \wedge z$	Φ_{12}: $w \wedge x$	Φ_{26}: $w \wedge x \wedge z$	Φ_{42}: *true*	Φ_{55}: $z \to a$
Φ_3 : $w \wedge z$	Φ_{14}: $x \wedge z$	Φ_{27}: $w \wedge x$	Φ_{43}: *true*	Φ_{56}: $a \wedge z$
Φ_4 : w	Φ_{15}: $x \wedge z$	Φ_{28}: $w \wedge x$	Φ_{46}: *true*	Φ_{59}: *true*
Φ_5 : w	Φ_{16}: z	Φ_{29}: w	Φ_{47}: *true*	Φ_{60}: *true*
Φ_6 : w	Φ_{17}: z	Φ_{30}: $w \to x$	Φ_{48}: w	Φ_{61}: a
Φ_7 : $w \to x$	Φ_{18}: $z \to x$	Φ_{31}: $w \wedge x$	Φ_{49}: w	Φ_{62}: a
Φ_9 : w	Φ_{19}: $z \to x$	Φ_{32}: $w \wedge x$	Φ_{52}: z	Φ_{63}: $a \wedge b$
Φ_{10}: $w \to x \wedge w \to z$	Φ_{20}: $x \wedge z$	Φ_{38}: *true*	Φ_{53}: z	Φ_{64}: $a \wedge b$
Φ_{11}: $w \to x$	Φ_{21}: $x \wedge z$	Φ_{39}: *true*	Φ_{54}: $z \to a$	

Note that the models $[\![\Phi_\ell]\!]$ coincide with $\mathcal{F}(\ell)$ from Ex. 4.2.

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

Given a program point ℓ, an MHF state Φ_ℓ, and an instruction to execute I_ℓ, our aim is to compute a new MHF state, denoted $\mu(I_\ell)$, that represents the effect of executing I_ℓ within Φ_ℓ. If I_ℓ is **skip**, then clearly $\mu(I_\ell) \equiv \Phi_\ell$. If I_ℓ is an **await** x? instruction, then $\mu(I_\ell)$ is $x \wedge \Phi_\ell$, which restricts the MHF

state of Φ_ℓ to those cases (i.e., models) in which x is finished. If I_ℓ is a call $y = m(\bar{x})$, where m is a method with parameters named $\bar{z}$, and, at the exit program point of m we know that the MHF state Φ_{ℓ_m} holds, then $\mu(I_\ell)$ is computed as follows:

- We compute an MHF state Φ_m that describes "what happens to tasks bound to $\bar{x}$ when m terminates". This is done by restricting Φ_{ℓ_m} to the method parameters, and then renaming the formal parameters $\bar{z}$ to the actual parameters $\bar{x}$, i.e., $\Phi_m = (\bar{\exists}\bar{z}.\Phi_{\ell_m})[\bar{z}/\bar{x}]$, where $[\bar{z}/\bar{x}]$ denotes the renaming.

- Now assume that ξ is a new (future) variable to which m is bound. Then $\xi \to \Phi_m$ states that "when m terminates, Φ_m must hold". Note that it says nothing about $\bar{x}$ if m has not terminated yet. It is also important to note that Φ_m is monotone so $\xi \to \Phi_m$ is a formula in our domain $\mathcal{A}_m$ (according to what is defined previously).

- Next we add $\xi \to \Phi_m$ to Φ_ℓ, eliminate (old) y since the variable is rewritten, and rename ξ to (new) y. Note that we use ξ as a temporary variable just not to conflict with the old value of y.

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

Example 4.4. *Let $\Phi_{11} = x \to w$ be the MHF state at program point 11. The effect of executing I_{11}, i.e., **await** w?, within Φ_{11} should eliminate all models that do not include w. This is done using $w \wedge \Phi_{11}$ which results in $\Phi_{12} = w \wedge x$. Now let $\Phi_{29} = w$ be the MHF state at program point 29. The effect of executing the instruction at program point 29, i.e., w=h(x,z), within Φ_{29} is defined as $\oplus(\Phi_{29}, w, \Phi_{56}, \langle x, z\rangle, \langle a, b\rangle)$ and is computed as follows:*

(1) we restrict $\Phi_{56} = a \wedge z$ to the method parameters $\langle a, b\rangle$, which results in a;

(2) we rename the formal parameters $\langle a, b\rangle$ to the actual ones $\langle x, z\rangle$ which results in $\Phi_h = x$;

(3) we compute $\exists w.(\Phi_{29} \wedge (\xi \to \Phi_h))$, which results in $\xi \to x$; and finally

(4) we rename ξ to w which results in $\Phi_{30} = \mathsf{w} \to \mathsf{x}$.

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

- if ℓ is not a method entry, we generate $\Phi_\ell = \vee\{\mu(\ell') \mid \ell' \in \texttt{pre}(\ell)\}$. This considers each program point ℓ' that immediately precedes ℓ, computes the effect $\mu(\ell')$ of executing $I_{\ell'}$ within $\Phi_{\ell'}$, and then takes their disjunction;
- if ℓ is an entry of method m, and $\ell_1, \ldots, \ell_k$ are the program points at which m is called, we generate $\Phi_\ell = \texttt{REACHED}(\Phi_{\ell_1}, \ldots, \Phi_{\ell_k}) \wedge (\wedge\{x \mid x \in L_m\})$. This means that all local variables point to finished tasks (since they are mapped to $\bot$ when entering a method), and we do not know anything about the parameters. In addition we require that m has been called – this is the role of $\texttt{REACHED}(\Phi_{\ell_1}, \ldots, \Phi_{\ell_k})$ which is a predicate that evaluates to *false* iff $\Phi_{\ell_1}, \ldots, \Phi_{\ell_k}$ are all *false*. Note that this predicate is *true* if m is a main method and thus we will ignore it in such case.

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

Example 4.5. *The equations for the program points of Figure 2.2 are:*

$$
\begin{array}{ll|ll}
\Phi_2 &= w \wedge x \wedge z & \Phi_{30} &= \oplus(\Phi_{29}, w, \Phi_{56}, \langle x, z\rangle, \langle a, b\rangle) \\
\Phi_3 &= \oplus(\Phi_2, y, \Phi_{39}, \langle\rangle, \langle\rangle) & \Phi_{31} &= w \wedge \Phi_{30} \\
\Phi_4 &= \oplus(\Phi_3, z, \Phi_{43}, \langle\rangle, \langle\rangle) & \Phi_{32} &= \Phi_{27} \\
\Phi_5 &= \Phi_4 & \Phi_{38} &= \texttt{REACHED}(\Phi_2, \Phi_{15}, \Phi_{26}) \\
\Phi_6 &= \Phi_5 & \Phi_{39} &= \Phi_{38} \\
\Phi_7 &= \oplus(\Phi_6, w, \Phi_{49}, \langle x\rangle, \langle w\rangle) & \Phi_{42} &= \texttt{REACHED}(\Phi_3, \Phi_{28}) \\
\Phi_9 &= \Phi_5 & \Phi_{43} &= \Phi_{42} \\
\Phi_{10} &= \oplus(\Phi_9, w, \Phi_{64}, \langle x, z\rangle, \langle a, b\rangle) & \Phi_{46} &= \texttt{REACHED}(\Phi_6, \Phi_{17}) \\
\Phi_{11} &= \Phi_7 \vee \Phi_{10} & \Phi_{47} &= \Phi_{46} \\
\Phi_{12} &= w \wedge \Phi_{11} & \Phi_{48} &= w \wedge \Phi_{47} \\
\Phi_{14} &= x \wedge z & \Phi_{49} &= \Phi_{48} \\
\Phi_{15} &= \Phi_{14} & \Phi_{52} &= \texttt{REACHED}(\Phi_{29}) \wedge z \\
\Phi_{16} &= \oplus(\Phi_{15}, x, \Phi_{39}, \langle\rangle, \langle\rangle) & \Phi_{53} &= \Phi_{52} \\
\Phi_{17} &= \Phi_{16} & \Phi_{54} &= \oplus(\Phi_{53}, z, \Phi_{49}, \langle a\rangle, \langle w\rangle) \\
\Phi_{18} &= \oplus(\Phi_{17}, z, \Phi_{49}, \langle x\rangle, \langle w\rangle) & \Phi_{55} &= \Phi_{54} \\
\Phi_{19} &= \Phi_{18} & \Phi_{56} &= z \wedge \Phi_{55} \\
\Phi_{20} &= z \wedge \Phi_{19} & \Phi_{59} &= \texttt{REACHED}(\Phi_9) \\
\Phi_{21} &= \Phi_{20} & \Phi_{60} &= \Phi_{59} \\
\Phi_{26} &= w \wedge x \wedge z & \Phi_{61} &= a \wedge \Phi_{60} \\
\Phi_{27} &= \oplus(\Phi_{26}, z, \Phi_{39}, \langle\rangle, \langle\rangle) \vee \Phi_{31} & \Phi_{62} &= \Phi_{61} \\
\Phi_{28} &= \Phi_{27} & \Phi_{63} &= b \wedge \Phi_{62} \\
\Phi_{29} &= \oplus(\Phi_{28}, x, \Phi_{43}, \langle\rangle, \langle\rangle) & \Phi_{64} &= \Phi_{63}
\end{array}
$$

Note the circular dependency of Φ_{27} and Φ_{31}. which originates from the corresponding ***while*** *loop. Recall that m_1,m_2,m_3 are main methods.*

The next step is to solve $\mathcal{H}_P$, i.e., compute an MHF state Φ_ℓ, for each $\ell \in \texttt{ppoints}(P)$, such that $\mathcal{H}_P$ is satisfiable. This can be done iteratively as follows. We start from an initial solution where $\Phi_\ell = \mathit{false}$ for each $\ell \in \texttt{ppoints}(P)$. Then repeat the following until a fixed-point is reached: (1) substitute the current solution in the right hand side of the equations, and obtain new values for each Φ_ℓ; and (2) merge the new and old values of each Φ_ℓ using $\vee$. E.g, solving the equation of Ex. 4.5 results in a solution that includes, among others, the MHF states of Ex. 4.3. In what follows we assume that $\mathcal{H}_P$ has been solved, and let Φ_ℓ be the MHF state at ℓ in such solution.

Theorem 4.1. *For any $\ell \in \texttt{ppoints}(P)$, we have $\mathcal{F}(\ell) \subseteq [\![\Phi_\ell]\!]$.*

The proof of this Theorem is given below. In the rest of this thesis we let $\texttt{mhf}_\alpha(\ell) = \{x \mid x \in V_m,\ \Phi_\ell \models x\}$, i.e., the set of finished future variables at ℓ that is induced by Φ_ℓ. Theorem 4.1 implies $\texttt{mhf}_\alpha(\ell) \subseteq \texttt{mhf}(\ell)$. Computing $\texttt{mhf}_\alpha(\ell)$ using the MHF states of Ex. 4.3, among others that are omitted, results exactly in the MHF sets of Ex. 4.1.

Proof of Theorem 4.1. The rest of this section proves Theorem 4.1. The proof follows the next general lines:

1. We first define a concrete *collecting semantics* that basically collects all reachable states when starting from some initial state;
2. We formulate of solving the MHF equations by menas of an abstract collecting semantics; and
3. We show that the abstract collecting semantics correctly approximates the concrete one with respect to the MHF property.

Concrete Collecting Semantincs. Recall that a program state S is a set of tasks (see Section 2.1). A concrete state in the collecting semantics, or simply *concrete state* or *state*, is a set of program states. We let $\mathcal{C}_0 = \{S_0\}$ where $S_0 = \{tsk(0, l, body(\mathsf{main}))\}$, i.e., an initial state that includes a single program state S_0 with a single task corresponding to method main. Let the function T be defined as follows

$$T = \lambda\mathcal{C}.\ \mathcal{C}_0 \cup \{S' \mid S \in \mathcal{C}, S \rightsquigarrow S'\}. \tag{4.1}$$

Then, the concrete collecting semantics is defined by

$$\mathcal{X} = lfp\, T. \tag{4.2}$$

Intuitively, $\mathcal{X}$ is the set of all reachable states when starting the execution from the initial state S_0.

Abstract Collecting Semantincs. Next we define the MHF analysis, i.e., solving the MHF equations, as an abstract collecting semantics. An abstract state Φ is a mapping from $\texttt{ppoints}(P)$ to Boolean formulas (as those defined in Section 4.2). The value to which ℓ is mapped in Φ is denoted by Φ_ℓ. Recall that our analysis is based on generating and solving a set of data-flow

equations $\mathcal{H}_P$. Given an abstract state Φ, we let $\mathcal{H}_P(\Phi)$ be the result of substituting the different formulas of Φ in the right-hand-side of $\mathcal{H}_P$, and in this way obtain new values for each Φ_ℓ. Given two abstract states Φ and Φ', we let $\Phi'' = \Phi \vee \Phi'$ be an abstract state such that $\Phi''_\ell = \Phi_\ell \vee \Phi'_\ell$ for any $\ell \in \texttt{ppoints}(P)$. We let Φ_0 be an initial abstract state such that $\Phi_\ell = \mathit{false}$ for each $\ell \in \texttt{ppoints}(P)$. Let the function $\bar{T}$ be defined as follows

$$\bar{T} = \lambda\Phi.\ \Phi_0 \vee \mathcal{H}_P(\Phi). \tag{4.3}$$

Then, the abstract collecting semantics, which is the result of our analysis, is defined by

$$\bar{\mathcal{X}} = \mathit{lfp}\,\bar{T}. \tag{4.4}$$

Next we define when an abstract state Φ correctly approximates a concrete one $\mathcal{C}$. Intuitively, by "correctly approximates" we mean that if in the concrete state we can reach a program point ℓ with an MHF set F, then this specific F is also present in the abstract state. The abstract state, however, might include spurious MHF sets that do not correspond to concrete one. First we recall the definition

$$f(S,l) = \{x \mid x \in \mathit{dom}(l),\ l(x) = \bot \vee (l(x) = \mathit{tid}' \wedge \mathit{tsk}(\mathit{tid}', l', \epsilon) \in S)\}$$

from Section 4.1, which is used to obtain the set of finished tasks, in a state S, for the future variables defined in l.

Definition 4.2. *An abstract state Φ correctly approximates a concrete state $\mathcal{C}$, denoted $\Phi \approx \mathcal{C}$, if for any $\mathit{tsk}(\mathit{tid}, l, s) \in S \in \mathcal{C}$, where $\mathit{pp}(s) = \ell$, it holds that $f(S,l) \in [\![\Phi_\ell]\!]$.*

The rest of this section shows that the abstract collecting semantics correctly approximates the concrete one, namely $\bar{\mathcal{X}} \approx \mathcal{X}$.

Lemma 4.1. $\forall n \geq 1.\ \exists k \geq n.\ \bar{T}^k(\Phi_0) \approx T^n(\emptyset)$.

Proof. The proof of the above Lemma is by induction on n.

For $n = 1$, we take $k = 1$, then $\mathcal{C}_0 = T(\emptyset)$ and $\Phi = \bar{T}(\Phi_0)$ is a state in which $\Phi_\ell = \mathit{false}$ for all $\ell \in \texttt{ppoints}(P)$ except for the entry point of main which is mapped to $\wedge\{x \mid x \in L_{\mathsf{main}}\}$. Clearly, $\Phi \approx \mathcal{C}_0$ as $\mathcal{C}_0 = \{S_0\} = \{\mathit{tsk}(0, l, \mathit{body}(\mathsf{main}))\}$ and $f(S_0, l) = \{x \mid x \in \mathit{dom}(l) = L_{\mathsf{main}}\}$.

For $n > 1$, let $\mathcal{C} = T^{n-1}(\emptyset)$ and $\mathcal{C}' = T^n(\emptyset)$. Note that $\mathcal{C}' = T(\mathcal{C})$. By the induction hypothesis, for there is $k \geq n-1$ such that $\Phi = \bar{T}^k(\Phi_0)$

correctly approximates $\mathcal{C}$. Let $S' \in \mathcal{C}'$ but $S' \notin \mathcal{C}$, and note that it must have been generated using some $S \in \mathcal{C}$ in one execution step, i.e., $S \rightsquigarrow S'$. In particular, by one execution step of a task $t = tsk(tid, l, b; s) \in S$, i.e., b was executed. Next we reason on all possible cases of b. First let us assume that $s \neq \epsilon$, i.e., this execution step does not introduce any new finished task, and later we come back to the case in which $s = \epsilon$. Assume that b corresponds to program point ℓ and that the first instruction in s corresponds to program point ℓ', i.e., $\ell \in \texttt{pre}(\ell')$.

Case 1: $b \equiv$ **skip.** In this case, the execution rewrites task $t = tsk(tid, l, \textsf{skip}; s)$ into $t' = tsk(tid, l, s)$ as well. Also in this case the status of each task $l(y)$, for any $y \in dom(l)$, in S' is the same as in S, since no finished task was introduced, i.e., $f(S', l) = f(S, l)$. Let $\Phi' = \bar{T}^{k+1}(\Phi_0)$, we claim that $f(S', l) \in [\![\Phi'_{\ell'}]\!]$. This is obvious since $f(S, l) \in [\![\Phi_\ell]\!]$ and $\Phi'_{\ell'}$ was obtained from an equation whose right-hand-side is a disjunction that includes $\mu(\Phi_\ell) = \Phi_\ell$.

Case 2: $b \equiv$ **await** $y?$**.** In this case, the execution rewrites task $t = tsk(tid, l, \textsf{await } y?; s)$ into $t' = tsk(tid, l, s)$. Note that the status of each task $l(y)$, for any $y \in dom(l)$, in S' is the same as in S, since no finished task was introduced ($l(y)$ must be finished in S to be able to execute this instruction), i.e., $f(S', l) = f(S, l)$. Let $\Phi' = \bar{T}^{k+1}(\Phi_0)$, we claim that $f(S', l) \in [\![\Phi'_{\ell'}]\!]$. This is obvious since $y \in f(S', l) = f(S, l) \in [\![\Phi_\ell]\!]$ and $\Phi'_{\ell'}$ was obtained from an equation whose right-hand-side is a disjunction that includes $\mu(\Phi_\ell) = \Phi_\ell \wedge y$. Adding y to Φ_ℓ does not eliminate models that include y, in particular $f(S', l)$.

Case 3: $b \equiv y = m(\bar{x})$**.** In this case, the execution rewrites task $t = tsk(tid, l, y = m(\bar{x}); s)$ into $t' = tsk(tid, l', s)$, and adds a new task $t'' = tsk(tid', l'', s')$ such that $l'(y) = tid'$. Let $\Phi' = \bar{T}^{k+1}(\Phi_0)$. From the same consideration as above it is easy to see that $f(S', l') \in [\![\Phi'_{\ell'}]\!]$ because the corresponding equation eliminates the old value of y from Φ_ℓ, and adds $y \rightarrow \Phi_m$. This implication has no effect when y is false. Let ℓ'' be the first program point of s', clearly $f(S', l'') \in [\![\Phi'_{\ell''}]\!]$ because of the way we generate the equation for entry program points (all local variables points to finished tasks, and parameters can be finished or not finished).

Now we go back to comment on the case that $s = \epsilon$. I.e., b was the

last instruction executed in task t. The problem here is that any other task that has a reference to t' must now consider the possibility that this task as finished. If we start from $\Phi' = \bar{T}^{k+1}(\Phi_0)$, and apply $\bar{T}$ one more time, we will reconsider the place where that method was called and use the new method summary Φ'_m, i.e., the method call will now add $y \to \Phi'_m$. This can be repeated a finite number of times until the information is propagated to all corresponding program points. □

Theorem 4.2 (Kleene fixed-point theorem)**.** *Let (L,⊑) be a CPO (complete partial order) and let $f : L \to L$ be a Scott-continuous (and therefore monotic) function. Then f has a least fixed point, which is the supremum of the ascending Kleene chain of f. That is*

$$lfp\ f = sup(\{f^n(\bot) \mid n \in (N)\})$$

Corollary 4.1. $\bar{\mathcal{X}} \approx \mathcal{X}$.

Proof. The proof is immediate from Lemma 4.1 and Kleene fixed-point theorem. For the concrete collecting semantics, we let the CPO L (of Theorem 4.2) be the set of all *concrete states* $\mathcal{C}$ with a partial order defined by the relation $\subseteq$ of sets, and the function f be T. For the abstract collecting semantics, we let the CPO L (of Theorem 4.2) be the set of *abstract states* Φ with a partial order defined by a logical implication, and the function f $\bar{T}$. It is obvious that T and $\bar{T}$ are Scott-continuous as they preserve all directed suprema. So we have the conditions to apply Theorem 4.2 and get:

$$\begin{aligned} \mathcal{X} &= lfp\, T = \quad sup(\{\, T^n(\emptyset) \mid n \in \mathbb{N}\}) \\ \bar{\mathcal{X}} &= lfp\, \bar{T} = \quad sup(\{\, \bar{T}^n(\Phi_0) \mid n \in \mathbb{N}\}) \end{aligned}$$

Then, by Lemma 4.1 we have $\bar{\mathcal{X}} \approx \mathcal{X}$. □

Chapter 5

MHP Analysis

In this chapter we present our MHP analysis, which is based on incorporating the MHF sets of Chapter 4 into the MHP analysis of Section 2.2. In sections 5.1 and 5.2 we describe how we modify the two phases of the original analysis, and describe the gain of precision with respect to Section 2.2 in each phase.

5.1 Local MHP

The local MHP analysis (LMHP) considers each method m separately, and for each $\ell \in \texttt{ppoints}(m)$ it infers an LMHP state that describes the tasks that might be executing when reaching ℓ (considering only tasks invoked in m). An LMHP state Ψ is a *multiset* of MHP atoms, where each atom represents a task and can be of the form:

(1) $y{:}\ell'{:}\tilde{m}(\bar{x})$, which represents an *active* task that might be at any of its program points, including the exit one, and is bound to future variable y. Moreover, this task is an instance of method m that was called at program point ℓ' (the *calling site*) with future parameters $\bar{x}$; or

(2) $y{:}\ell'{:}\hat{m}(\bar{x})$, which differs from the previous one in that the task can only be at the exit program point, i.e., it is a *finished* task.

In both cases, future variables y and $\bar{x}$ can be $\star$, which is a special symbol indicating that we have no information on the future variable.

Intuitively, the MHP atoms of Ψ represent (local) tasks that are executing in parallel. However, since a variable y cannot be bound to more than one

task at the same time, atoms bound to the same variable represent *mutually exclusive tasks*, i.e., cannot be executing at the same time. The same holds for atoms that use *mutually exclusive calling sites* ℓ_1 and ℓ_2 (i.e., there is no path from ℓ_1 to ℓ_2 and vice versa). The use of multisets allows including the same atom several times to represent different instances of the same method. We let $(a, i) \in \Psi$ indicate that a appears i times in Ψ. Note that i can be ∞, which happens when the atom corresponds to a calling site inside a loop, this guarantees convergence of the analysis. Note also that the MHP atoms of Section 2.2 do not use the parameters $\bar{x}$ and the calling site ℓ', since they do not benefit from such extra information.

Example 5.1. *The following are LMHP states for some program points from Figure 2.2:*

$L2$: $\{\}$
$L3$: $\{x{:}2{:}\tilde{f}()\}$
$L4$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}()\}$
$L5$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}()\}$
$L6$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}()\}$
$L7$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}(), w{:}6{:}\tilde{g}(x)\}$
$L8$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}(), w{:}6{:}\tilde{g}(x)\}$
$L9$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}(), w{:}6{:}\tilde{g}(x)\}$
$L10$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}(), w{:}9{:}\tilde{k}(x, z)\}$
$L11$: $\{x{:}2{:}\tilde{f}(), z{:}3{:}\tilde{q}(), w{:}6{:}\tilde{g}(x), w{:}9{:}\tilde{k}(x, z)\}$
$L12$: $\{x{:}2{:}\hat{f}(), z{:}3{:}\tilde{q}(), w{:}6{:}\hat{g}(x), w{:}9{:}\hat{k}(x, z)\}$
$L14$: $\{\}$
$L15$: $\{\}$
$L16$: $\{x{:}15{:}\tilde{f}()\}$
$L17$: $\{x{:}15{:}\tilde{f}()\}$
$L18$: $\{x{:}15{:}\tilde{f}(), z{:}17{:}\tilde{g}(x)\}$
$L19$: $\{x{:}15{:}\tilde{f}(), z{:}17{:}\tilde{g}(x)\}$
$L20$: $\{x{:}15{:}\hat{f}(), z{:}17{:}\hat{g}(x)\}$
$L21$: $\{x{:}15{:}\hat{f}(), z{:}17{:}\hat{g}(x)\}$
$L26$: $\{\}$
$L27$: $\{z{:}26{:}\tilde{f}(), (\star{:}28{:}\hat{q}(), \infty), (\star{:}29{:}\hat{h}(\star, z), \infty)\}$
$L28$: $\{z{:}26{:}\tilde{f}(), (\star{:}28{:}\hat{q}(), \infty), (\star{:}29{:}\hat{h}(\star, z), \infty)\}$
$L29$: $L27 \cup \{x{:}28{:}\tilde{q}()\}$
$L30$: $L29 \cup \{w{:}29{:}\tilde{h}(x, z)\}$
$L31$: $L27$
$L32$: $L27$
$L38$: $\{\}$
$L39$: $\{\}$
$L42$: $\{\}$
$L43$: $\{\}$
$L46$: $\{\}$
$L47$: $\{\}$
$L48$: $\{\}$
$L49$: $\{\}$
$L52$: $\{\}$
$L53$: $\{\}$
$L54$: $\{z{:}53{:}\tilde{g}(a)\}$
$L55$: $\{z{:}53{:}\tilde{g}(a)\}$
$L56$: $\{z{:}53{:}\hat{g}(a)\}$
$L59$: $\{\}$
$L60$: $\{\}$
$L61$: $\{\}$
$L62$: $\{\}$
$L63$: $\{\}$
$L64$: $\{\}$

Let us explain some of the above LMHP states. The state at L5 includes $x{:}2{:}\tilde{f}()$ *and* $z{:}3{:}\tilde{q}()$ *for the active tasks invoked at L2 and L3. The state at*

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

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

- if I_ℓ is a call $y = m(\bar{x})$, then $\tau(I_\ell) = \Psi_\ell[y/\star] \cup \{y{:}\ell'{:}\tilde{m}(\bar{x})\}$, which replaces each occurrence of y by $\star$, since it is rewritten, and then adds a new atom $y{:}\ell{:}\tilde{m}(\bar{x})$ for the newly created task. E.g., the LMHP state of L30 in Ex. 5.1 is obtained from the one of L29 by adding w:29:$\tilde{h}$(x, z) for the call at L29;
- if I_ℓ is **await** $y?$, and ℓ' is the program point after ℓ, then we mark all tasks that are bound to a finished future variable as finished, i.e., $\tau(I_\ell)$ is obtained by turning each $z{:}\ell''{:}\tilde{m}(\bar{x}) \in \Psi_\ell$ to $z{:}\ell''{:}\hat{m}(\bar{x})$ for each $z \in \mathtt{mhf}_\alpha(\ell')$. E.g., the LMHP state of L12 in Ex. 5.1 is obtained from the one of L11 by turning the status of g, k, and f to finished (since w and x are finished at L12);
- otherwise, $\tau(I_\ell) = \Psi_\ell$.

The main difference w.r.t. the analysis of Section 2.2 is the treatment of **await** $y?$: while we use an MHF set computed using the inter-procedural MHF analysis of Section 4, In Section 2.2 the MHF set $\{y\}$ is used, which is obtained syntactically from the instruction. Our LMHP analysis, as in Section 2.2, is defined as a solution of a set of LMHP constraints. In what follows we assume that the results of the LMHP analysis are available, and we will refer to the LMHP state of program point ℓ as Ψ_ℓ.

5.2 Global MHP

The results of the LMHP analysis are used to construct an MHP graph, from which we can compute the desired set of MHP pairs. The construction is exactly as in Section 2.2 except that we carry the new information in the MHP atoms. However, the process of extracting the MHP pairs from such graphs will be modified.

In what follows, we use $y{:}\ell{:}\breve{m}(\bar{x})$ to refer to an MHP atom without specifying if it corresponds to an active or finished task, i.e., the symbol $\breve{m}$ can be matched to $\tilde{m}$ or $\hat{m}$. As in Section 2.2, the nodes of the MHP graph consist of two method nodes $\tilde{m}$ and $\hat{m}$ for each method m, and a program point node ℓ for each $\ell \in \texttt{ppoints}(P)$. Edges from $\tilde{m}$ to each $\ell \in \texttt{ppoints}(m)$ indicate that when m is active, it can be executing at any program point, including the exit, but only one. An edge from $\hat{m}$ to ℓ_m indicates that when m is finished it can be only at its exit program point. The out-going edges from a program point node ℓ reflect the atoms of the LMHP state Ψ_ℓ as follows: if $(y{:}\ell'{:}\breve{m}(\bar{x}), i) \in \Psi_\ell$, then there is an edge from node ℓ to node $\breve{m}$ and it is labeled with $i{:}y{:}\ell'{:}\bar{x}$. These edges simply indicate which tasks might be executing in parallel when reaching ℓ, exactly as Ψ_ℓ does.

Example 5.2. *The MHP graphs $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ in Figure 3.1, Figure 3.2 and Figure 3.3 correspond to methods m_1, m_2, and m_3, each analyzed together with its reachable methods. For simplicity, the graphs include only some program points of interest. Note that the out-going edges of program point nodes coincide with the LMHP states of Ex. 5.1.*

The procedure of Section 2.2 for extracting the MHP pairs from the (modified) MHP graph of a program P, denoted $\mathcal{G}_P$, is based on the following principle: (ℓ_1, ℓ_2) is an MHP pair induced by $\mathcal{G}_P$ iff

(i) $\ell_1 \rightsquigarrow \ell_2 \in \mathcal{G}_P$ or $\ell_2 \rightsquigarrow \ell_1 \in \mathcal{G}_P$; or

(ii) there is a program point node ℓ_3 and paths $\ell_3 \rightsquigarrow \ell_1 \in \mathcal{G}_P$ and $\ell_3 \rightsquigarrow \ell_2 \in \mathcal{G}_P$, such that the first edges of these paths are different and they do not correspond to mutually exclusive MHP atoms, i.e., they use different future variables and do not correspond to mutually exclusive calling sites (see Section 5.1). Edges with multiplicity $i > 1$ represent i different edges.

The first (resp. second) case is called direct (resp. indirect) MHP, see Chapter 3.

Example 5.3. *Let us explain some of the MHP pairs induced by* $\mathcal{G}_1$ *of Figure 3.1. Since* $11 \rightsquigarrow 38 \in \mathcal{G}_1$ *and* $11 \rightsquigarrow 42 \in \mathcal{G}_1$*, we conclude that (11,42) and (11,38) are direct MHP pairs. Moreover, since these paths originate in the same node 11, and the first edges use different future variables, we conclude that (42,38) is an indirect MHP pair. Similarly, since* $11 \rightsquigarrow 46 \in \mathcal{G}_1$ *and* $11 \rightsquigarrow 59 \in \mathcal{G}_1$ *we conclude that (11,46) and (11,59) are direct MHP pairs. However, in this case (46,59) is not an indirect MHP pair because the first edges of these paths use the same future variable* w*. Indeed, the calls to* g *and* k *appear in different branches of an* **if** *statement. To see the improvement w.r.t. to Section 2.2 note that node 12 does not have an edge to* $\tilde{f}$*, since our MHF analysis infers that* x *is finished at that L12. The analysis of Section 2.2 would have an edge to* $\tilde{f}$ *instead of* $\hat{f}$*, and thus it produces spurious pairs such as (12,38). Similar improvements occur also in* $\mathcal{G}_2$ *and* $\mathcal{G}_3$*.*

Now consider nodes 38 and 48, and note that we have $11 \rightsquigarrow 38 \in \mathcal{G}_1$ *and* $11 \rightsquigarrow 48 \in \mathcal{G}_1$*, and moreover these paths use different future variables. Thus, we conclude that (38,48) is an indirect MHP pair. However, carefully looking at the program we can see that this is a spurious pair, because* x *(to which task* f *is bound) is passed to method* g*, as parameter* w*, and* w *is guaranteed to finish when executing* **await** w? *at L47. A similar behavior occurs also in* $\mathcal{G}_2$ *and* $\mathcal{G}_3$*. For example, the paths* $30 \rightsquigarrow 42 \in \mathcal{G}_3$ *and* $30 \rightsquigarrow 48 \in \mathcal{G}_3$ *induce the indirect MHP pair (42,48), which is spurious since* x *is passed to* h *at L29, as parameter* a*, which in turn is passed to* g *at L53, as parameter* w*, and* w *is guaranteed to finish when executing* **await** w? *at L47.*

The spurious pairs in the above example show that even if we used our improved LMHP analysis when constructing the MHP graph, using the procedure of Section 2.2 to extract MHP pairs might produce spurious pairs. Next we handle this imprecision, by modifying the process of extracting the MHP pairs to have an extra condition to eliminate such spurious MHP pairs. This condition is based on identifying, for a given path $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$, which of the parameters of m are guaranteed to finish before reaching ℓ, and thus, any task that is passed to m in those parameters cannot execute in parallel with ℓ.

Definition 5.1. *Let* p *be a path* $\breve{m} \rightsquigarrow \ell \in \mathcal{G}_P$*,* $\bar{z}$ *be the formal parameter of* m*, and* I *a set of parameter indices of method* m*. We say that* I *is* not alive along p *if*

(i) p *has a single edge, and for some* $i \in I$ *the parameter* z_i *is in* $\texttt{mhf}_\alpha(\ell)$*; or*

(ii) p *is of the form* $\breve{m} \longrightarrow \ell_1 \overset{k:y:\ell':\bar{x}}{\longrightarrow} \breve{m}_1 \rightsquigarrow \ell$, *and for some* $i \in I$ *the parameter* z_i *is in* $\mathtt{mhf}_\alpha(\ell_1)$ *or* $I' = \{j \mid i \in I, z_i = x_j\}$ *is not alive along* $\breve{m}_1 \rightsquigarrow \ell$.

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

Example 5.4. *Consider* $p \equiv \tilde{g} \rightsquigarrow 48 \in \mathcal{G}_1$, *and let* $I = \{1\}$, *then* I *is not alive along* p *since it is a path that consists of a single edge and* $\mathsf{w} \in \mathtt{mhf}_\alpha(48)$. *Now consider* $\tilde{h} \rightsquigarrow 48 \in \mathcal{G}_3$, *and let* $I = \{1\}$, *then* I *is not alive along* p *since* $I' = \{1\}$ *is not alive along* $\tilde{g} \rightsquigarrow 48$.

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

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

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

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

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

Example 5.5. *We reconsider the spurious indirect MHP pairs of Ex. 5.3. Consider first (38,48), which originates from*

$$p_1 \equiv 11 \overset{1:\mathsf{x}:2:[]}{\longrightarrow} \tilde{f} \rightsquigarrow 38 \in \mathcal{G}_1 \text{ and } p_2 \equiv 11 \overset{1:\mathsf{w}:6:[\mathsf{x}]}{\longrightarrow} \tilde{g} \rightsquigarrow 48.$$

We have $F = \{\mathsf{x}, \mathsf{w}\}$, $I = \{1\}$, *and we have seen in Ex. 5.4 that* I *is not alive along* $\tilde{g} \rightsquigarrow 48 \in \mathcal{G}_1$, *thus* p_1 *and* p_2 *are mutually exclusive and we eliminate this pair. Similarly, consider (42,48) which originates from*

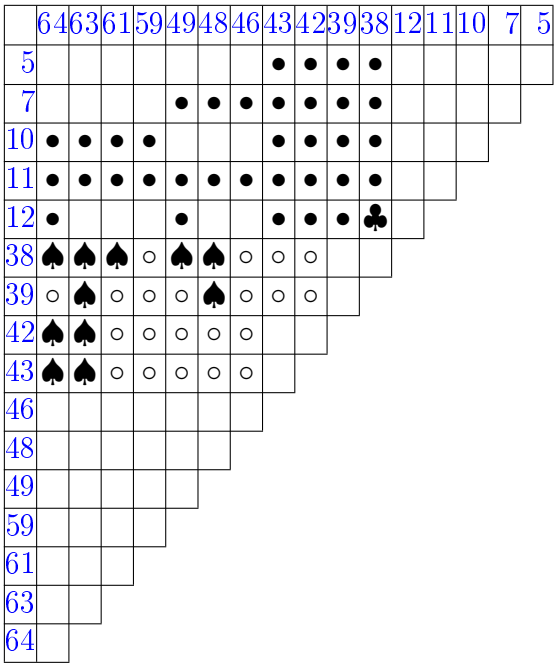

Figure 5.1: MHP pairs from m_1

$p_1 \equiv 30 \xrightarrow{1:x:28:[]} \tilde{q} \rightsquigarrow 42 \in \mathcal{G}_3$ *and* $p_2 \equiv 30 \xrightarrow{1:w:29:[x,z]} \tilde{h} \rightsquigarrow 48$. *Again* $F = \{x, w\}$, $I = \{1\}$, *and we have seen in Ex. 5.4 that* I *is not alive along* $\tilde{h} \rightsquigarrow 48 \in \mathcal{G}_3$, *thus* p_1 *and* p_2 *are mutually exclusive and we eliminate this pair.*

Example 5.6. *In Figure 5.1 and 5.2 are shown all MHP pairs from methods* m_1,m_2 *and* m_3. *Note that they are main methods. In each table we can distinguish between different types of pairs. If the cell that connects two nodes is marked with* • *indicates that the pair is a direct MHP. Cells marked with* ∘ *indicate that the pair is a indirect MHP. Cells marked with* ♣ *or* ♠ *represent spurious pairs that the analysis described in Section 2.2 will infer. Cells marked with* ♣ *represent the relation* $y \to x$. *Cells marked with* ♠ *represent that a task is awaited inside other method. The tables represent the* $\tilde{\mathcal{E}}_P$ *obtained from the graphs of Figure 3.2, Figure 3.1 and Figure 3.3. They also capture the MHP relations informally discussed in Section 2.1 and 3.*

Let $\tilde{\mathcal{E}}_P$ be the set of all MHP pairs obtained by applying the process of

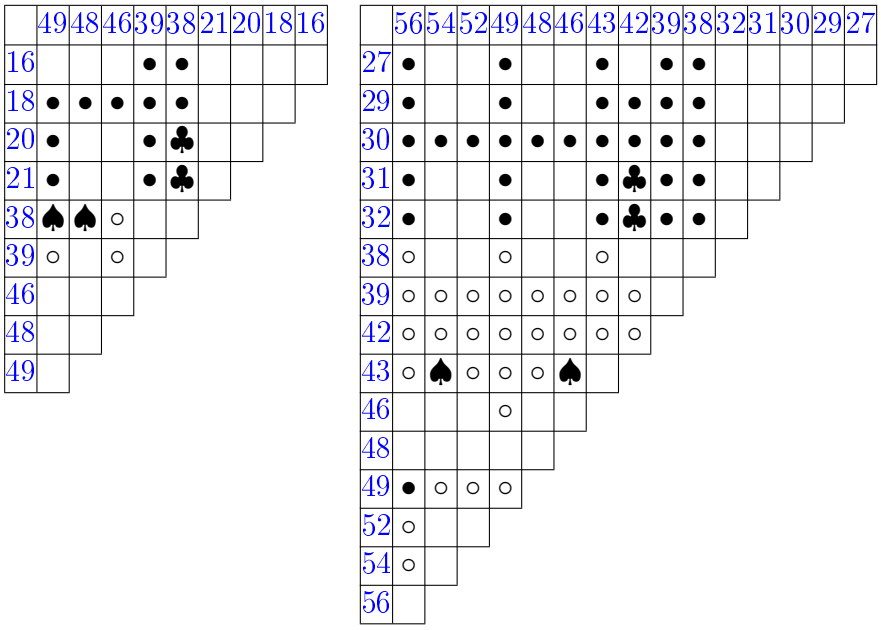

Figure 5.2: (LEFT) MHP pairs from m_2. (RIGHT) MHP pairs from m_3.

Section 2.2, modified to eliminate indirect pairs that correspond to mutually exclusive paths.

Theorem 5.1. $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

Note that a proof for Theorem 5.1 is not given since it is basically as the one of [3] but (1) using the MHF information, for the case of **await** y? in the local analysis, which is straightforward; and (2) using the *mutually exclusive* path condition whose correctness was intuitively argued in Section 5.1 already.

Chapter 6

Implementation

SACO [2] is a *S*tatic *A*nalyzer for *C*oncurrent *O*bjects, which is able to infer deadlock, termination and resource boundedness of ABS [12], a distributed asynchronous language based on concurrent objects. Concurrent objects are based on the notion of concurrently running objects, similar to the actor-based and active-objects approaches [14, 15]. These models take advantage of the inherent concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way.

The MHF analysis has been implemented and its output has been used within the local and global phases of the MHP analysis, which have been adapted to this new input. Although our analysis has been formalized for the abstract model described in Section 2.1, our implementation is done for the full concurrent object-oriented language ABS [12].

The information can be displayed by means of a graphical representation of the MHP analysis graph or in a textual way, as a set of pairs which identify the program points that may run in parallel. The set can be obtained for all program points, or only for those ones of interest, or on demand.

The web interface of SACO can bi tried online at: `http://costa.ls.fi.upm.es/saco/web`. To use it, it is necessary to enable the option `Inter-Procedural Synchronization` of the `MHP` analysis within the `Settings` in the menu. One can then apply the MHP analysis by selecting it from the menu for the desired type of analyses and then clicking on `Apply`.

6.1 Implementation Details

This section reports on some of the real implementation details.

First, the representation of the analysis and the graph is presented. After that, the implementation of the MHF analysis are presented with its main operations and how the MHP graph changes and is built. Finally we present how to get the MHP pairs.

The mhp atoms which refer to the same method m are represented as a tuple of the form $<\mathbb{P}_m,\mathbb{A}_m,\mathbb{E}_m,p_m,a_m,e_m>$ where:

- $\mathbb{P}_m,\mathbb{A}_m,\mathbb{E}_m$ are arrays of size $futvar_m$, the number of future variables in the method that is being analyzed. Their domain is dom(P) = dom(A) = dom(E) = B^n where B $\in \{0,1\}$.

  Assuming that future variables are enumerated : $\mathbb{P}$[i] = 1 $\Leftrightarrow$ y_i:$\check{m}$; $\mathbb{A}$[i] = 1 $\Leftrightarrow$ y_i:$\tilde{m}$; $\mathbb{E}$[i] = 1 $\Leftrightarrow$ y_i:$\hat{m}$.

- p_m,a_m,e_m are natural numbers or infinite. Their values represent the following: if p = i $\Leftrightarrow$ $(\star{:}\check{m}, i) \in M$; a = i $\Leftrightarrow$ $(\star{:}\tilde{m}, i) \in M$; e = i $\Leftrightarrow$ $(\star{:}\hat{m}, i) \in M$ where i $\in \mathbb{N} \cup \infty$.

The result of the MHF analysis is represented by means of tuples of the form $<name_m,id_m,fut_m,\mathbb{C}_m>$ where:

- $name_m$ is a chain of characters that represents the name of a method m $\in$ `methods`(P) and id_m is its identifier (which is unique). Their domains are dom($name_m$) = B^n where B $\in \{a,...,z\}$ and dom(id_m) = $\mathbb{N}$.

- fut_m is a set of future variables that must have finished when the task m finishes. Its elements are of the form y:[$\mathbb{D}_m$] where $\mathbb{D}_m$ represents the future variables whose tasks are ensured to be finished if the task associated to y has finished. Their domains are dom(y) = L_m and dom($\mathbb{D}_m$) = P_m.

- $\mathbb{C}_m$ is a list of the methods that are called inside $body(m)$. Its domain is dom($\mathbb{C}_m$) = `methods`(P).

The MHP graph $\mathcal{G}_P$ = (V,E) is a direct graph with a set of nodes V and a set of edges E. The nodes are represented as tuples $<name_V,id_V>$ where:

- $name_V$ is a tuple which allows distinguishing between the different types of nodes (see Section 2.2.3). These tuples have the state of the task (active,pending,finished) or if the program point is the entry or the exit point of the method, respectively, and the name of the method which the node refers to.

- id_V is a numeric value that identifies uniquely each node. Its domain is $\text{dom}(id_V) = \mathbb{N}$.

On the other hand, the edges are also represented as tuples which can be of two forms, $<S,T,W>$ or $<S,T,W,y,fut>$ where:

- S and T identify the source node and the target node of the edge respectively. Their domain is $\text{dom}(S)=\text{dom}(T)= \mathbb{N}$.

- W is the multiplicity of the instances of the method, that the edge is related to, that might be running in parallel when reaching the program point of the node S. Its domain is $\text{dom}(W)=\mathbb{N} \cup \infty$.

- y refers to the future variable associated to the method m that is represented by node T. Its domain is $\text{dom}(y) = L_m$.

- fut is a list which contains the future variables that are ensured to be finished if the task m associated to y has finished. It is obtained as the result of MHF analysis. Its domain is $\text{dom}(fut) = P_m$.

6.2 Computation of MHF Analysis

First, program P is analyzed in order to build the set of all equations $\mathcal{H}_P$ in Section 4. It is implemented as a set of dependences between each method and its calls. The result of each equation that has been solved is substituted in all places where it appears. This process is done iteratively until all dependences are solved.

There is a tuple $<name_m,id_m,fut_m,\mathbb{C}_m>$ for each method m $\in$ `methods`(P). For a method m, the set fut_m is initialized to the future variables whose associated task is guaranteed to be finished (those that appear in an **await** ? statement) and the set $\mathbb{C}_m$ to the methods that are asynchronously called in its *body*. Then, in an iterative way, the set fut_m of the methods whose computation has finished is combined with the set of those methods that call

m. The computation of a method has finished if it does not have any call to other methods or if it has calls but the computation of the called methods has finished too.

In the case that method m has a conditional instruction, it has to wait until both branches of the conditional statement finish their computation. After that, their results are combined to get the correct one.

When this process finishes, the set fut_m of each tuple has all the future variables that will finish when the method m finishes. To create the structure described in the previous section and represent the relation $\rightarrow$ between the future variables, each tuple search which of the future variables that are in its set fut_m are passed as arguments to the methods that appear in $\mathbb{C}_m$. Those ones have been included as a result of the MHF analysis. The termination of the tasks associated to these futures depends on the termination of the method in $\mathbb{C}_m$ that contains them as its arguments. When it is found, these future variables and the local future variable on which depends are deleted from fut_m and the structure $y : [\mathbb{D}]$ is included.

6.3 Computation of LMHP Analysis

To compute the LMHP analysis, each method is analyzed separately until it reaches a fixed-point. First, for each method m of program P, its body is analyzed instruction per instruction. Each of these instructions modifies the tuple $<\mathbb{P}_m,\mathbb{A}_m,\mathbb{E}_m,p_m,a_m,e_m>$. If the instruction is a method call, it checks if the future variable associated to the method called has already been used. It also checks if the object that calls the method is blocked to include the future variable as pending. In case that the instruction is an await statement, it changes the positions associated to the future variable of $\mathbb{A}_m$ to 0 and put the correct one in $\mathbb{E}_m$. It also activates all the tasks that are pending by modifying $\mathbb{P}_m$ and $\mathbb{A}_m$.

When the analysis of method m finishes, the old tuple must be updated with the new results if it has changed. To verify this, the values of p_m,a_m,e_m are compared with the old values in order to know if a new instance of a method has been created (the multiplicity of the method has increased). It also checks if the state of each future variable has changed.

The execution continues with the computation of the application-level analysis when the fixed-point is reached. The result of the LMHP analysis and the MHF analysis will be used to create the graph.

6.4 MHP Graph

To build the MHP graph $\mathcal{G}_P$, first the nodes are created. For each method m of the program P, the nodes that represent the three states that method m can have (active,pending or finish) and those that represent its program points are created. This process is done using the MHP tuples $<\mathbb{P}_m,\mathbb{A}_m,\mathbb{E}_m,p_m,a_m,e_m>$ of each method m of P. The nodes associated to each future variable are created too. After that, the edges are built using the MHP atoms as they contain all the information needed.

When all the edges have been created according to Section 2.2, the result of MHF analysis is used to modify them and include the additional information.

It is done checking if the edges or the nodes have been created. If not, they are created.

After building the graph, what remains is to search the MHP pairs. In order to get them, the nodes which are reachable, from each one are computed. Then, we can infer the direct MHP pairs. To infer the indirect MHP pairs, it is only necessary to look for paths that have the same source node and check if these paths satisfy the conditions (see Section 5.2) not to discard these pairs.

6.5 Graphical Interface

We have integrated the implementation of our analysis in the graphical interface of SACO usable from `http://costa.ls.fi.upm.es/saco/web`.

In Figure 6.1 we show a view of the tool with the running example *interproc_2.abs* that can be found in the folder "SAS15 Interproc. MHP Example". It is a translation of the method m_2 in Figure 2.2 to ABS.

To enable our analysis we have to click on the Settings button that appears in the top of Figure 6.1. The settings menu is shown in Figure 6.2. The intra-procedural MHP analysis will be executed by default. To execute our analysis, it is necessary to click on the check button "Inter-Procedural Synchronization" shown in Figure 6.2.

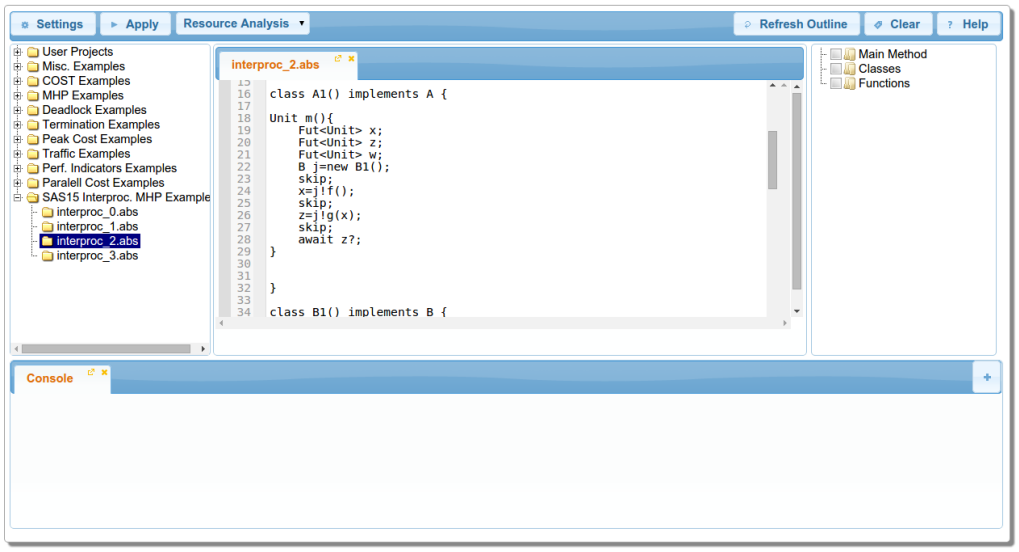

Figure 6.1: Translation of method m_2 in language ABS.

One can then apply the MHP analysis by selecting it from the menu for the type desired analysis. Before clicking on Apply, you have to click on Refresh and select the check button "Main Method".

In the bottom of Figure 6.3 we show the result of applying the MHP analysis with Inter-procedural Synchronization to the method shown in Figure 6.1. We have two ways to see the result. In the Figure 6.4, we can see information about the analysis such as the time spent in executing each part of the analysis and the result in text format. It shows pairs of program points that can run in parallel.

In Figure 6.5 we show the other alternative to see the result of the analysis. If you click on the arrows that appear in the left of the lines, those that can run in parallel with the selected line get highlighted.

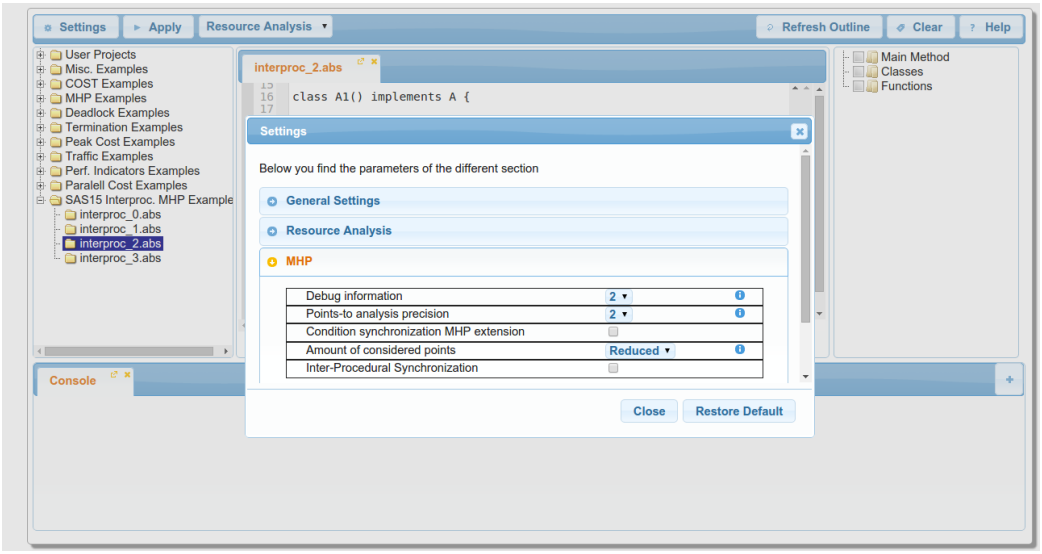

Figure 6.2: Settings Menu of SACO.

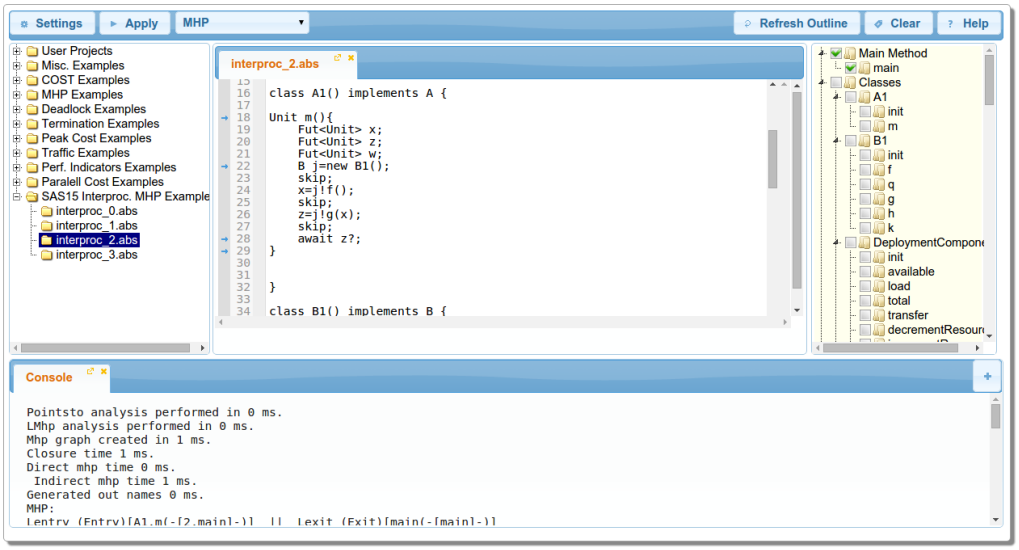

Figure 6.3: Result of executing example interproc_2.abs.

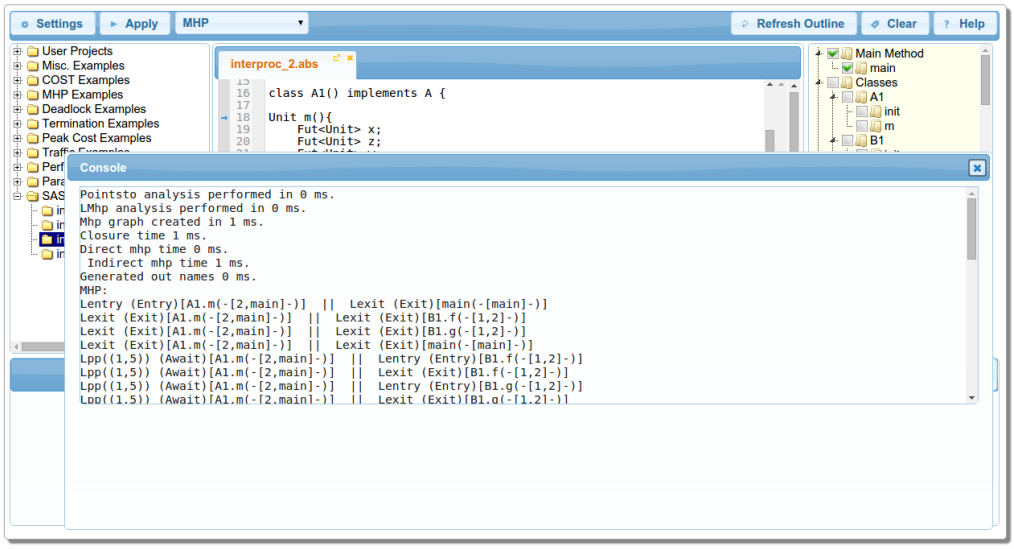

Figure 6.4: Result of our analysis in text mode.

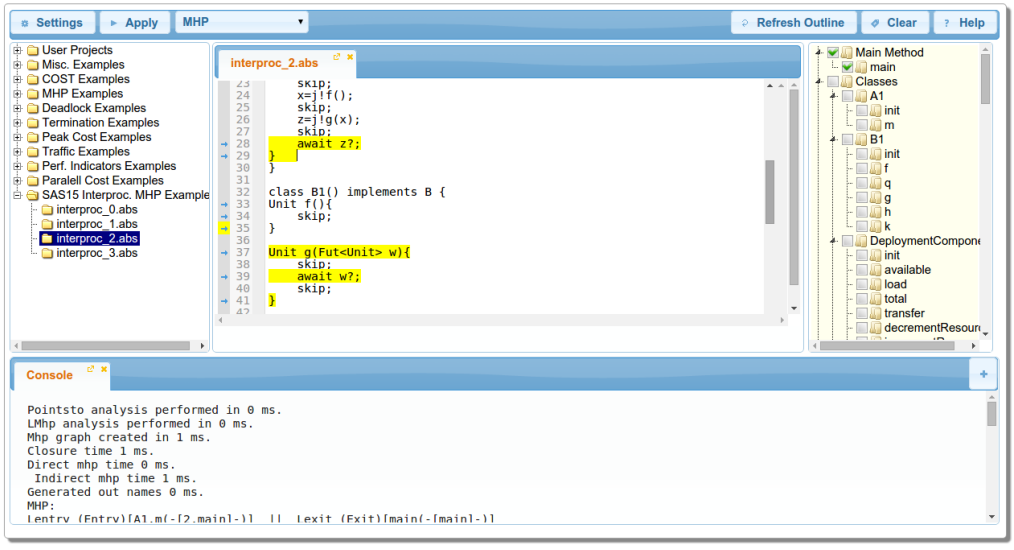

Figure 6.5: Result of our analysis in visual mode.

Chapter 7

Conclusions, Related and Future Work

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

7.1 Related Work

There is an increasing interest in asynchronous programming and in concurrent objects, and in the development of program analyses that reason on safety and liveness properties [7]. The authors of [7] have developed an approximation to refinement checking. They propose a characterization of observational refinement as a set-inclusion problem defined independently from execution contexts of libraries. It is done defining partial orders between the operations which are admitted by each library. The criterion of observational refinement is formalized using labeled transition systems (LTS). Finally, they reduce refinement checking to safety-property checking using symbolic arithmetic representations.

Existing MHP analyses for asynchronous programs [3, 13, 1] lose all infor-

mation when future variables are used as parameters, as they do not handle inter-procedural synchronization. In [13, 1], the problem of developing a MHP analysis is studied for the concurrency model with "async-finish parallelism". The "async" construct allows forking a process and creating new threads and the "finish" construct ensures that all methods called within its scope terminate before the execution continues. In [1], the authors propose a MHP analysis in two steps. First, it is computed a Never-Execute-in-Parallel (NEP) analysis (complement of MHP) to discard some pairs. Then, a Place-Equivalent analysis is performed to know if all instances of two statements are guaranteed to execute at the same places. Finally, both analyses are combined to obtain MHP information. To compute that, they use a program structure tree to represent each procedure, in contrast to us, that we use a graph that represents the whole program analyzed. [13] presents a MHP analysis of a storeless model of X10. The authors focus on answering two problems closely related: the MHP decision problem and the MHP computation problem. The first one will be used to answer the second. To solve the first problem they use a reduction to constrained dynamic pushdown networks (CDPNs). CDPN models collections of sequential pushdown processes running in parallel. They give a translation from programs in X10 to CDPNs and the MHP decision problem is solved by performing a reachability test. To solve the MHP computation problem a type-based analysis that produces a set of candidate pairs is developed. The type analysis problem is recasted as a constraint solving problem. Once this set has been created, the CDPN-based decision procedure is used to each of the candidate pairs in order to remove those that cannot happen in parallel. Our analysis also solves both problems of [13] using the MHP graph. To answer the decision problem, we have only to compute if there is a path between the nodes that represents the two program point and, in such case, check if it holds the condition.

As a consequence, existing analysis for more advanced properties [11, 4] that rely on the MHP relations lose the associated analysis information on such futures.

In [11] a MHP analysis based on deadlock is presented. This analysis is done in two steps. First it is defined the notion of deadlock based in the extended one of [8]. What the authors define are state dependences. Then, an abstract dependency graph is built with them. After that, cycles are searched in order to know if there is a deadlock. When every cycle has been declared, in a post-process, MHP analysis is used to eliminate unfeasible scenarios in which the cycles are built with program points that cannot run in parallel.

Here, the problem is that neither the state dependences nor MHP analysis consider inter-procedural synchronization. Despite that, [11] extends its basic framework to handle future variables stored in fields. Something similar happens in [4]. It presents a termination and cost analysis. Its reasoning is at the level of strongly connected components. The idea is to assume a property (finiteness) on the global state to prove termination of a loop and then prove that this property holds. In order to prove that a program P terminates, it is proved that all its strongly connected components terminate. The authors have developed an algorithm that is able to do that. However, it is necessary the information inferred by a MHP analysis and it does not handle inter-procedural synchronization. This implies that the result of the analysis may be imprecise if future variables are passed as arguments to methods.

7.2 Future Work

In the near future, we plan to apply our analysis to industrial case studies that are being developed in ABS but that are not ready for experimentation yet.

In addition, we plan to study the computational complexity of deciding MHP, for our abstract model, with and without inter-procedural synchronizations in a similar way to what has been done in [6] for the problem of state reachability.

Most of existing MHP analyses lose the inter-procedural information as they do not support it and do not treat future variables. We want to continue studying the use of future variables and its variants. We plan to enhance the existing MHP analyses by handly methods that return future variables or have future variables as fields.

Bibliography

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In K. A. Yelick and J. M. Mellor-Crummey, editors, *Proc. of PPOPP'07*, pages 183–193. ACM, 2007.

[2] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.

[3] E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.

[4] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. Termination and Cost Analysis of Loops with Concurrent Interleavings. In *ATVA 2013*, LNCS 8172, pages 349–364. Springer, October 2013.

[5] Elvira Albert, Samir Genaim, and Pablo Gordillo. 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, 2015. To appear.

[6] A. Bouajjani and M. Emmi. Analysis of Recursively Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(3):10, 2013.

[7] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Tractable Refinement Checking for Concurrent Objects. In *Proc. of POPL 2015*, pages 651–662. ACM, 2015.

[8] F. S. de Boer, M. Bravetti, I. Grabe, M. David Lee, M. Steffen, and G. Zavattaro. A Petri Net based Analysis of Deadlocks for Active Objects and Futures. In *Proc. of FACS 2012*, 2012.

[9] F. S. de Boer, D. Clarke, and E. B. Johnsen. A Complete Guide to the Future. In *ESOP'07*, LNCS 4421, pages 316–330. Springer, 2007.

[10] C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimization. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[11] A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, LNCS, pages 273–288. Springer, 2013.

[12] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10 (Revised Papers)*, LNCS 6957, pp. 142-164. Springer, 2012.

[13] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient May Happen in Parallel Analysis for Async-Finish Parallelism. In *In SAS 2012*, volume 7460, pages 5–23. Springer, 2012.

[14] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *Proc. of ECOOP'10*, LNCS, pages 275–299. Springer, 2010.

[15] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proc. of ECOOP'08*, LNCS 5142, pages 104–128. Springer, 2008.