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

Antonio E. Flores Montoya

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Programación y Tecnología Software

21 de Junio de 2012

Director/es y/o colaborador:

Elvira Albert Albiol
Samir Genaim

Convocatoria: Junio de 2012
Calificación obtenida: Sobresaliente

Autorización de difusión

Antonio E. Flores Montoya

21 de Junio de 2012

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Analysis of May-Happen-in-Parallel in Concurrent Objects", realizado durante el curso académico 2011-2012 bajo la dirección de Elvira Albert Albiol y Samir Genaim en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Firmado: Antonio E. Flores Montoya

Resumen en castellano

Esta tesis presenta un análisis *may-happen-in-parallel* (puede-ocurrir-en-paralelo) para lenguajes orientados a objetos y basados en *objetos concurrentes*. En éste modelo de concurrencia, los objetos son la *unidad* de concurrencia tal que, cuando un método de un objeto o_2 es llamado desde una tarea que se ejecuta en un objeto o_1, las instrucciones de la tarea concurrente en o_1 pueden ejecutarse en paralelo con las de la llamada (asíncrona) en o_2, y con los métodos llamados de forma transitiva. El objetivo del análisis MHP es identificar los pares de instrucciones que puede ejecutarse en paralelo en un programa para cualquier ejecución del mismo. Nuestro análisis está definido como un análisis a nivel de método (*local*) cuya información se puede componer de forma modular para obtener la información al nivel de la aplicación (*global*). Se ha demostrado la corrección del análisis, es decir, que sobre-aproxima el paralelismo real de los programas de acuerdo con una semántica bien definida. Se ha implementado un prototipo, dando lugar al sistema MayPar, un analizador may-happen-in-parallel que se puede ser útil para depuración, corrección de errores y para incrementar la precisión de otros análisis que infieran propiedades más complejas (por ejemplo, terminación o análisis de recursos). MayPar se ha probado con múltiples ejemplos, entre ellos dos casos de estudio industriales, y se han alcanzado buenas cotas de precisión y rendimiento.

Palabras clave

May-Happen-in-Parallel, Análisis Estático, Sistemas Asíncronos Distribuidos, Objetos Concurrentes, Actors, Lenguaje ABS

Abstract

This thesis presents a *may-happen-in-parallel* (MHP) analysis for object-oriented languages based on *concurrent objects*. In this concurrency model, objects are the concurrency *units* such that, when a method is invoked on an object o_2 from a task executing on object o_1, the statements of the current task in o_1 may run in parallel with those of the (asynchronous) call on o_2, and with those of transitively invoked methods. The goal of the MHP analysis is to identify pairs of statements in the program that may run in parallel in any execution. Our MHP analysis is formalized as a method-level (*local*) analysis whose information can be modularly composed to obtain application-level (*global*) information. The analysis has been proven to be sound, that is, it over-approximates the real parallelism according to a well defined semantics. A prototype implementation has been carried out, resulting in the MayPar system, a may-happen-in-parallel analyzer that can be used for debugging and to increase precision of other analyses which infer more complex properties (e.g., termination and resource consumption). MayPar has been tested on several examples, including two industrial cases studies, achieving good precision and performance.

Keywords

May-Happen-in-Parallel, Static analysis, Distributed Asynchronous Systems, Concurrent Objects, Actors, ABS Language

Table of Contents

Acknowledgments **iii**

1 Introduction **1**

1.1 Concurrent Objects 2

1.2 May-Happen-in-Parallel analysis 3

1.3 Contributions 5

2 Concurrent Objects **6**

2.1 Operational Semantics 7

3 Definition of MHP **11**

3.1 MHP examples 12

4 MHP Analysis **16**

4.1 Inference of method-level MHP 16

4.2 The Notion of MHP Graph 20

4.3 Inference of Global MHP 23

4.4 Partial analyses and points of interest 26

4.5 Soundness and Complexity 27

5 MayPar Tool: Implementation and Experimental Evaluation **29**

5.1 Implementation Details 29

5.2 Method-level analysis state representation and operations 30

5.2.1 State Representation 30

5.2.2 Inclusion operation 30

5.2.3 Upper bound operation 32

5.2.4 Optimizations 33

5.3 Use of the Tool 33

5.3.1 The Notifier Example 34

5.3.2 MHP Analysis of the Notifier example 35

5.3.3 Using MHP for debugging and understanding concurrent programs . 37

5.3.4 Using MHP information in static program analysis 38

5.4 Experimental Evaluation 39

6 Conclusions, Related and Future Work **41**
6.1 Related Work . . . 41
6.2 Future Work . . . 42

Bibliography **44**

A Proofs **47**
A.1 Proof of Corollary 4.4.2 . . . 47
A.1.1 Proof of theorem 4.5.1 . . . 47
A.2 Proof of Theorem A.1.4 . . . 50
A.3 Proof of Theorem A.1.7 . . . 55
A.4 Proof of Theorem A.2.1 . . . 57
A.5 Proof of Theorem A.3.2 . . . 57

Acknowledgements

I am very grateful to my supervisors Elvira and Samir for their invaluable help and support. I want to thank Elvira for her positive attitude and her practical point of view kept my feet on the ground and helped me see the big picture and Samir for his insightful advices and expertise that were essential for the success of this thesis and his constant desire to help. The professors of the research master enriched my experience as well. The problems and techniques they presented inspired me in more that one occasion and broadened my perspective. Last but not least, I want to express my gratitude to my family for their love and unconditional support.

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

Chapter 1

Introduction

Concurrent applications have been thoroughly studied for a long time now. Their importance and use has grown further with the success of parallel architectures and this tendency is expected to continue and become even more apparent in the future.

These kind of applications can no longer be conceived as a lineal sequence of instructions executed one after another. In concurrent applications, several tasks or processes run in parallel. Multiple execution flows progress at different speeds, they collaborate, synchronize, coordinate and even compete with each other. All these interactions result in a dynamic and complex environment with many possible behaviors and great doses of non-determinism. System managers, programmers and software analysts entangled in the design of these systems can easily end overwhelmed by their complexities. Reasoning about concurrent systems is very difficult. Testing and debugging can also be really hard as, very often, bugs cannot be easily reproduced. Thus, well defined methodologies and formal methods can play a capital role in producing robust and high quality systems.

With the widespread of high speed connections and the Internet, distributed applications have also gained much relevance. Distributed applications not only have multiple components progressing at the same time, but also these components can be executed in different locations far away from each other. This delocalization makes communication and synchronization even more challenging. Over the Internet, delays are hardly predictable and the possibility of failure is always present. As a result, some paradigms, like the ones based on shared memory, cannot be easily applied.

Furthermore, models based in shared memory, locks and threads (such as Java) lead to a low level programming style, which have shown to be error prone and, more importantly, not *modular* enough.

1.1 Concurrent Objects

The actor concurrency model [2] has lately regained attention for the design of distributed applications. In this model, a system is composed by a series of entities called actors. Each actor has a state and can communicate with other actors by means of asynchronous messages (which fits well in a distributed environment). In the initial model, an actor can react to a message just by sending new messages, changing its internal state or creating new actors. The order in which the messages are received is not determined beforehand and actors cannot modify other actor's state directly.

This model has been successfully adopted by programming languages like Erlang [7] which was especially designed for distributed applications. There are also several libraries for implementing actors in other languages (such as the actors library in SCALA [11] and Java [20]).

It is easy to see that Actors and object oriented paradigms have many concepts in common such as data ownership and encapsulation. This makes these two approaches suitable to be combined. The concurrent objects model is based in the Actor model but is perfectly integrated in the object oriented paradigm. Instead of having asynchronous messages, concurrent objects adopt a similar approach based on asynchronous method calls and future variables [10]. An asynchronous method call works similarly to an asynchronous message where the called method is the type of message sent and the arguments are the passed information. However, asynchronous method calls provide an additional functionality supported by the future variables. When a method is called, it can be associated to a future variable. In such case, once the method has finished the return value is stored in the future variable and can be retrieved by the caller. Additionally, a method can use a future variable to synchronize and wait until its associated call has finished. In particular, consider an asynchronous method call m on object o, written as f=o.m(). f is the future variable and the instruction **await** f? allows checking whether m has finished, and let the current task release the processor to allow another available task to take it. In addition, the instruction *get.f* is used to retrieve the result of the asynchronous call. If the result is not available yet, it blocks the execution on the current object until the call finishes.

Each object is considered as a concurrency unit, that is, each object conceptually has a dedicated processor. A method call creates a new task inside the target object. This task will share a single processor with every other task in the same object.

An essential feature of this paradigm is that task scheduling in *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This enables reasoning about the object state and combining passive and active behavior. This

cooperative strategy together with the fact that objects' data is always private results in a data-race free language (i.e. fields cannot be read or modified by to tasks at the same time). It is easier to prevent bad behaviors provoked by unexpected interleavings as interleaving points are explicit. Objects internally behave similarly to monitors where the lock is given by the semantics or to thread pools with restricted interleaving.

Actors and shared memory models are not the only existing concurrency models. There are models based on synchronous message passing, such as *rendevouz* implemented in languages like ADA. In this paradigm both sides of communication have to be ready to exchange messages. This can lead to much undesired waiting in a distributed system where elements are away from each other.

Other languages like X10 have a different concurrency model called **async-finish** parallelism. This model is based in two primitives: **async**, which triggers the execution of a new task; and **finish**, that waits until all tasks started in its scope have finished.

1.2 May-Happen-in-Parallel analysis

This thesis proposes a *may-happen-in-parallel* (MHP) analysis for concurrent objects. The goal of an MHP analysis is to identify pairs of statements that can execute in parallel (see, e.g., [13]).

In the context of concurrent objects, an asynchronous method invocation $\mathsf{f{=}o_2.m();}$ within a task t_1 executing in an object o_1 implies that the subsequent instructions of t_1 in o_1 may execute in parallel with the instructions of m within o_2. However, if the asynchronous call is synchronized with an instruction **await** f?, after executing such an *await*, it is ensured that the execution of the call to m has terminated and hence, the instructions after the **await** cannot execute in parallel with those of m. Inferring precise MHP information is challenging because, not only does the current task executes in parallel with m, but also with other tasks that are *transitively* invoked from m. Besides, two tasks can execute in parallel even if they do not have a transitive invocation relation. For instance, if we add an instruction $\mathsf{f_3{=}o_3.p();}$ below the previous asynchronous invocation to m in t_1, then instructions in p may run in parallel with those of m. This is a form of *indirect* MHP relation in which tasks run in parallel because they have a common ancestor. The challenge is to precisely capture in the analysis all possible forms of MHP relations.

Computing the exact MHP information is undecidable and in practice, performing an analysis that considers all possible interleavings would be unfeasible as the number of possibilities grows exponentially. Thus, it is necessary to safely approximate all these possible behaviors. A May-Happen-in-Parallel analysis can restrict the number of possibilities and,

consequently, allow other analyses to obtain better precision at a lower cost. That is, it can act as a starting point to later construct different kinds of verification and testing tools which build on it in order to infer more complex properties. For example, in order to prove termination (or infer the cost) of a simple loop of the form **while** (l!=**null**) {f=o.process(l. data); **await** f?; l=l.next;}, assuming l is a shared variable (i.e., field), we need to know the tasks that can run in parallel with the body of the loop to check whether the length of the list l can be modified during the execution of the loop by some other task when the processor is released (at the **await**). For concurrent languages which are not data-race free, MHP is fundamental in order to verify the absence of data-races.

On the other hand, it provides very useful information to automatically extract the maximal level of parallelism for a program and improve performance. In the context of concurrent objects, when the methods running on two different objects may run in parallel, it can be profitable to deploy such objects on different machines in order to improve the overall performance.

The information provided by this analysis can also be used directly by the programmer in order to identify bugs. If the analysis reports that two code fragments can run in parallel and they were not supposed to (or vice versa), it is possible that there is an error in the design or in the implementation. Furthermore, by a closer inspection (see chapter 5) this kind of analysis can help the programmer locate the source of the error. In general, the MHP analysis can help the designer to have a better understanding of the program behavior so he can see whether it matches its intended behavior or not. This way, he can identify possible flaws or defects in both the implementation and design.

The nature of a MHP analysis is intrinsically determined by the targeted concurrency model. Several MHP analysis have been developed but none of them correspond to the concurrent objects model. There are many analyses that approximate the MHP information for "rendevouz" style languages such like ADA [18]. However, it is not clear up to what point it is possible to apply these concepts to concurrent objects as their behavior is essentially different. There are also MHP analyses for Java programs [19, 16, 9]. but these face additional problems derived from the low level characteristics of its concurrency model. Objects and threads need to be treated independently and often it is necessary to infer which objects are used in different threads. Recently, X10 and its concurrency model async-finish have been the target of MHP analyses [13, 1]. The async-finish parallelism is more restrictive than other models thanks to the finish primitive which guarantees that everything within its scope has finished. This restriction leads to more efficient MHP analyses, however, it makes the language less interesting from the programmer's point of view.

1.3 Contributions

The main contributions of this thesis are:

- The proposal is, to the best of our knowledge, the first MHP analysis for concurrent objects. The analysis has two main phases:
    1. *method-level* MHP information is inferred by locally analyzing each method and ignoring transitive calls. This is a new local analysis which, among other things, collects the *escape* points of method calls, i.e., those program points in which the asynchronous calls terminate but there might be transitive asynchronous calls not finished.
    2. The other new aspect is that the method-level information is modularly composed in order to obtain *application-level* (*global*) MHP information. The composition is achieved by constructing an *MHP analysis graph* which over-approximates the parallelism –both implicit and through transitive calls– in the application. Then, the problem of inferring if two statements x and y can run in parallel amounts to checking certain *reachability* conditions between x and y in the MHP analysis graph.
- We have developed *MayPar*, a standalone application that integrates the analysis for the ABS language. ABS [12] is a actor-based language which has been recently proposed to model distributed concurrent objects. The application has a web interface available at `http://costa.ls.fi.upm.es/costabs/mhp`. It can be used for program verification, debugging purposes and its results can be easily interpreted by other applications. The implementation has been evaluated on small applications which are classical examples of concurrent programming and on two industrial case studies. Results on the efficiency and accuracy of the analysis are promising.
- The analysis has also been integrated in COSTABS [3], a cost and termination analyzer for the ABS language.

The main technical results have been published in the proceedings of FMOODS&FORTE 2012 [5] and a tool demo description of *MayPar* has been submitted to FSE 2012 [6].

Chapter 2

Concurrent Objects

We describe the syntax and semantics of the simple imperative language with concurrent objects on which we develop our analysis. It is basically the subset of the ABS language [12] relevant to the MHP analysis. Class, method, field, and variable names are taken from a set $\mathcal{X}$ of valid *identifiers*. A *program* consists of a set of classes $\mathcal{K} \subseteq \mathcal{X}$. The set *Types* is the set of possible types $\mathcal{K} \cup \{\textsf{int}\}$, and the set $\mathit{Types}_{\mathcal{F}}$ is the set of future variable types defined as $\{Fut\langle t\rangle \mid t \in \mathit{Types}\}$. A *class declaration* takes the form:

$$\textsf{class}\ \kappa_1\ \{t_1\ \mathit{fn}_1;\ldots\ t_n\ \mathit{fn}_n;\ M_1\ \ldots\ M_k\}$$

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

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

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

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

There is an implicit local variable called **this** that refers to the current object. x and y represent variables of types $t \in \mathit{Types}$ and $ft \in \mathit{Types}_{\mathcal{F}}$ respectively. Observe that only fields of the current object **this** can be accessed (this, together with the semantics, make the language be data-race free [12]). We assume the program includes a method called main

without parameters, which does not belong to any class and has no fields, from which the execution will start.

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

Without loss of generality, we assume that all methods in the program have different names. As notation, we use $body(m)$ for the sequence of instructions defining method m, $P_{\mathcal{M}}$ for the set of method names defined in a program P, $P_{\mathcal{F}}$ for the set of future variable names defined in a program P.

2.1 Operational Semantics

A program state S is a tuple $S = \langle O, T\rangle$ where O is a set of objects and T is a set of tasks. Only one task can be *active* (running) in each object and has the object's *lock*. All other tasks are *pending* to be executed or *finished* if they terminated and released the lock.

The set of objects O includes all available objects. An object takes the form $\langle oid, lk, f\rangle$ where oid is a unique identifier taken from an infinite set of identifiers $\mathcal{O}$, $lk \in \{\top, \bot\}$ indicates whether the object's lock is free ($\top$) or not ($\bot$), and $f : \mathcal{X} \rightarrow \mathcal{O} \cup \mathbb{Z} \cup \{\textsf{null}\}$ is a partial mapping from object fields to values.

The set of tasks T represents those tasks that are being executed. Each task takes the form $\langle tid, m, oid, lk, l, s\rangle$ where tid is a unique identifier of the task taken from an infinite set of identifiers $\mathcal{T}$, m is the method name executing in the task, oid identifies the object to which the task belongs, $lk \in \{\top, \bot\}$ is a flag that indicates if the task has the object's lock or not, $l : \mathcal{X} \rightarrow \mathcal{O} \cup \mathcal{T} \cup \mathbb{Z} \cup \{\textsf{null}\}$ is a partial mapping from local (possibly future) variables to their values, and s is the sequence of instructions still to be executed. Given a task tid, we assume that $object(tid)$ returns the object identifier oid of the corresponding task.

The execution of a program starts from the initial state:

$$S_0 = \langle\{\langle 0, \bot, f\rangle\}, \{\langle 0, \textsf{main}, 0, \top, l, body(\textsf{main})\rangle\}\rangle$$

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

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

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

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

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

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

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

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

Figure 2.1: *Semantics*

Where f is an empty mapping (since main had no fields), and l maps local references and future variables to **null** and integer variables to 0.

The execution proceeds from S_0 by applying *non-deterministically* the semantic rules depicted in Fig. 2.1. We use the notation $\{t \parallel T\}$ to represent that task t is non-deterministically selected for execution.

The operational semantics is given in a rewriting-based style where at each step a subset of the state is rewritten according to the rules as follows:

1. executes an instruction in a task that has its object lock. These instructions may change the heap (global state), the local state and the sequence of instructions that

$$eval(x{=}e; s, O, l, oid) = \quad (O, l', s) \qquad l' = l[x \rightarrow value(e, l, f_{oid})]$$

$$eval(\textsf{this}.fn{=}e; s, O, l, oid) = (O', l, s) \qquad \begin{array}{r} O' = O[\langle oid, \bot, f_{oid}\rangle / \langle oid, \bot, f'_{oid}\rangle] \\ f'_{oid} = f[fn \rightarrow value(e, l, f_{oid})] \end{array}$$

$$eval(x{=}\textsf{new}\ \kappa(\bar{x}); s, O, l, oid) = (O', l', s) \qquad \begin{array}{r} O' = O \cup \{\langle oid', \top, f\rangle\} \\ oid' \text{ is a fresh id} \\ f = init(\bar{x}, l, f_{oid}) \\ l' = l[x \rightarrow oid'] \end{array}$$

$$\begin{array}{ll} eval(\textsf{ if } b \textsf{ then } s_1 \textsf{ else } s_2; s, O, l, oid) = & \\ \qquad (O, l, s_1; s) & if\ value(e, l, f_{oid}) = true \\ \qquad (O, l, s_2; s) & otherwise \end{array}$$

$$\begin{array}{ll} eval(\textsf{while } b \textsf{ do } s_3; s, O, l, oid) = & \\ \qquad (O, l, s_3; \textsf{while } b \textsf{ do } s_3; s) & if\ value(e, l, f_{oid}) = true \\ \qquad (O, l, s) & otherwise \end{array}$$

Figure 2.2: eval *function represents the semantics of the sequential instructions*

are left to execute (in the case of an if-then-else or a while instruction). Such changes are captured in function *eval* specified at Fig. 2.2.

2. A method call creates a new task (the default values of the local variables are set by *buildLocals*) with a fresh task identifier which is associated to the corresponding future variable.

3. When **return** is executed, the return value is stored in v so that it can be obtained by the future variables that point to that task. Besides, the lock is released and will never be taken again by that task (the notation $O[o/o']$ is used to replace o by o' in O). Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$), though it does not disappear as other tasks might need to access its return value.

4. If the future variable we are awaiting for points to a finished task, the await can be completed.

5. The await can be substituted by a release plus an await. This allows us to await until rule (4) can be applied.

6. A task executes a release and yields the lock so that any other task of the same object can take it.

7. A non finished task can obtain its object lock if it is unlocked.

8. A y.**get** instruction waits for the future variable but without yielding the lock. It then retrieves the value associated with the future variable y. Note that this instruction would automatically produce a deadlock if the corresponding future variable y is associated to a method from the same object.

Chapter 3

Definition of MHP

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

Definition 3.0.1 (Runtime MHP)**.** *Given a program P, we let $\mathcal{E}^r_P = \cup\{\mathcal{E}^r_S \mid S_0 \leadsto^* S\}$ where for the state $S = \langle O, Tk\rangle$, the set $\mathcal{E}^r_S$ is defined as $\mathcal{E}^r_S = \{((tid_1, pp(s_1)), (tid_2, pp(s_2))) \mid \langle tid_1, m_1, oid_1, lk_1, l_1, s_1\rangle \in Tk, \langle tid_2, m_2, oid_2, lk_2, l_2, s_2\rangle \in Tk, tid_1 \neq tid_2\}$.*

$\mathcal{E}^r_P$ defines the set of pairs of specific tasks in a certain point of their execution (at a program point) that can belong to a single program state. This definition takes into account all reachable configurations during runtime. However, as we ignore which tasks are going to be created at runtime, we can abstract away from task identifiers.

Definition 3.0.2 (MHP)**.** *Given a program P, its MHP is defined as $\mathcal{E}_P = \cup\{\mathcal{E}_S | S_0 \leadsto^* S\}$ where for the state $S = \langle O, Tk\rangle$, the set $\mathcal{E}_S$ is defined as $\mathcal{E}_S = \{(pp(s_1),\ pp(s_2)) \mid \langle tid_1, m_1, o_1, lk_1, l_1, s_1\rangle \in Tk, \langle tid_2, m_2, o_2, lk_2, l_2, s_2\rangle \in Tk, tid_1 \neq tid_2\}$.*

$\mathcal{E}_P$ can be seen as an abstraction of $\mathcal{E}^r_P$ where task ids are ignored and we just keep the information related to program points. $\mathcal{E}_P$ is the information we want to approximate.

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

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

Figure 3.1: *Simple examples for different MHP behaviours.*

An important characteristic of this definition is that having $((tid_1, pp(s_1)), (tid_2, pp(s_2))) \in \mathcal{E}_P^r$ does not imply that both task tid_1 and tid_2 can progress at that point. In particular, if both tasks belong to the same object at most one of them can have the lock. This does not suppose a limitation in practice. In case we were interested in the parallelism information within an object (where this situation can happen), only release points, where interleavings can occur, are relevant to us. But at the release points, the definition works as expected because the object lock is free and can be obtained by any task of the object.

3.1 MHP examples

Let us explain first the notions of *direct* and *indirect* MHP and *escaped* methods, which are implicit in the definition of MHP above, on the simple representative patterns in Fig. 3.1.

There are 4 versions of m which use the methods p, q and r. Only the parts of p and q useful for explaining the MHP behavior are shown (the code of r is irrelevant). However, we assume that r does not call any other method and it has two program points $p_{\mathring{r}} = L60$ and $p_{\check{r}} = L61$. We also assume that the last instruction of each method is a **return** and that m belongs to a different object than p and q. The global MHP behavior of executing each m (separately) is as follows.

A p and q are called from m, then r is called from p and q. The await instruction in program point 6 (L6 for short) ensures that q will have finished afterwards. If q has finished executing, its call to r has to be finished as well because there is an await in L47. The await instruction in L8 waits until p has finished before continuing. That means that at L9, p is not longer executing. However, the call to r from p might be

still executing. We say that r might *escape* from p. Method calls that might escape need to be considered.

The following table shows the complete $\mathcal{E}_P$ relation for example A. Instead of enumerating all pairs and for readability, for each program point p, $\mathcal{E}_P(p)$ express the set of all program points that can happen in parallel with p.

$$
\begin{array}{rcl}
\mathcal{E}_P(1) = \mathcal{E}_P(2) = \mathcal{E}_P(3) & = & \emptyset \\
\mathcal{E}_P(4) & = & \{41-44, 60, 61\} \\
\mathcal{E}_P(5) = \mathcal{E}_P(6) & = & \{41-50, 60, 61\} \\
\mathcal{E}_P(7) = \mathcal{E}_P(8) & = & \{41-44, 50, 60, 61\} \\
\mathcal{E}_P(9) = \mathcal{E}_P(10) & = & \{44, 50, 60, 61\} \\
\mathcal{E}_P(41) & = & \{4-8, 45-50, 60, 61\} \\
\mathcal{E}_P(42) = \mathcal{E}_P(43) & = & \{4-8, 45, 47, 50, 60, 61\} \\
\mathcal{E}_P(44) & = & \{4-10, 45-50, 60, 61\} \\
\mathcal{E}_P(45) = \mathcal{E}_P(47) & = & \{5, 6, 41-44, 60, 61\} \\
\mathcal{E}_P(46) = \mathcal{E}_P(48) = \mathcal{E}_P(49) & = & \{5, 6, 41, 44, 60, 61\} \\
\mathcal{E}_P(50) = & = & \{5-10, 41-44, 60, 61\} \\
\mathcal{E}_P(60) = \mathcal{E}_P(61) & = & \{4-10, 41-50, 60, 61\}
\end{array}
$$

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

The complete $\mathcal{E}_P$ is the following:

$$
\begin{array}{rcl}
\mathcal{E}_P(11) = \mathcal{E}_P(12) = \mathcal{E}_P(13) & = & \emptyset \\
\mathcal{E}_P(14) & = & \{60\} \\
\mathcal{E}_P(15) = \mathcal{E}_P(16) & = & \{41-44, 60, 61\} \\
\mathcal{E}_P(17) & = & \{41-50, 60, 61\} \\
\mathcal{E}_P(18) = \mathcal{E}_P(19) = \mathcal{E}_P(20) & = & \{41-44, 50, 60, 61\} \\
\mathcal{E}_P(41) = \mathcal{E}_P(42) & = & \{15-20, 41-50, 60, 61\} \\
\mathcal{E}_P(43) = \mathcal{E}_P(44) & = & \{15-20, 41-50, 60, 61\} \\
\mathcal{E}_P(45) = \mathcal{E}_P(47) & = & \{17, 41-44, 60, 61\} \\
\mathcal{E}_P(46) = \mathcal{E}_P(48) = \mathcal{E}_P(49) & = & \{17, 41, 44, 60, 61\} \\
\mathcal{E}_P(50) & = & \{17-20, 41-44, 60, 61\} \\
\mathcal{E}_P(60) = & = & \{14-20, 41-50, 60, 61\} \\
\mathcal{E}_P(61) & = & \{15-20, 41-50, 60, 61\}
\end{array}
$$

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

The complete $\mathcal{E}_P$ is the following:

$$
\begin{array}{rcl}
\mathcal{E}_P(21) = \mathcal{E}_P(22) & = & \emptyset \\
\mathcal{E}_P(23) = \mathcal{E}_P(24) = \mathcal{E}_P(26) & = & \{41-44, 50, 60, 61\} \\
\mathcal{E}_P(27) = \mathcal{E}_P(28) = \mathcal{E}_P(29) & = & \{41-44, 50, 60, 61\} \\
\mathcal{E}_P(30) & = & \{41-44, 50, 60, 61\} \\
\mathcal{E}_P(25) & = & \{41-44, 50, 60, 61\} \\
\mathcal{E}_P(41) = \mathcal{E}_P(44) & = & \{23-30, 41-44, 45-50, 60, 61\} \\
\mathcal{E}_P(42) = \mathcal{E}_P(43) & = & \{23-30, 41, 44, 45, 47, 50, 60, 61\} \\
\mathcal{E}_P(45) = \mathcal{E}_P(47) & = & \{25, 41-44, 50, 60, 61\} \\
\mathcal{E}_P(46) = \mathcal{E}_P(48) = \mathcal{E}_P(49) & = & \{25, 41, 44, 50, 60, 61\} \\
\mathcal{E}_P(50) & = & \{23-30, 41-50, 60, 61\} \\
\mathcal{E}_P(60) = \mathcal{E}_P(61) & = & \{23-30, 41-50, 60, 61\}
\end{array}
$$

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

The complete $\mathcal{E}_P$ is the following:

$$
\begin{aligned}
\mathcal{E}_P(31) = \mathcal{E}_P(32) = \mathcal{E}_P(33) &= \emptyset \\
\mathcal{E}_P(34) = \mathcal{E}_P(36) &= \emptyset \\
\mathcal{E}_P(37) = \mathcal{E}_P(38) &= \{41-50, 60, 61\} \\
\mathcal{E}_P(39) = \mathcal{E}_P(40) &= \{44, 50, 60, 61\} \\
\mathcal{E}_P(41) = \mathcal{E}_P(42) &= \{37, 38\} \\
\mathcal{E}_P(43) &= \{37, 38, 60, 61\} \\
\mathcal{E}_P(44) &= \{37-40, 60, 61\} \\
\mathcal{E}_P(45) = \mathcal{E}_P(46) &= \{37, 38\} \\
\mathcal{E}_P(47) &= \{37, 38, 60, 61\} \\
\mathcal{E}_P(48) = \mathcal{E}_P(49) &= \{37, 38, 61\} \\
\mathcal{E}_P(50) &= \{37-40, 61\} \\
\mathcal{E}_P(60) &= \{37-40, 43, 44, 47\} \\
\mathcal{E}_P(61) &= \{37-40, 43, 44, 47-50\}
\end{aligned}
$$

Chapter 4

MHP Analysis

The problem of inferring $\mathcal{E}_P$ is clearly undecidable in our setting [14], and thus we develop a MHP analysis which statically approximates $\mathcal{E}_P$. The analysis is done in two main steps, first it infers method-level MHP information. Then, in order to obtain application-level MHP, it composes this information by building a MHP graph whose paths provide the required global MHP information.

4.1 Inference of method-level MHP

The method-level MHP analysis is used to infer the local effect of each method on the global MHP property. In particular, for each method m, it infers, for each program point $p \in m$, the status of all tasks that (might) have been invoked (within m) so far.

The status of a task can be (1) *pending*, which means that it has been invoked but has not started to execute yet, i.e., it is at the entry program point; (2) *finished*, which means that it has finished executing already, i.e., it is at the exit program point; and (3) *active*, which means that it can be executing at any program point (including the entry and the exit). As we explain later, the distinction between these statuses is essential for precision.

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

Example 4.1.1. *Consider programs A and B in Fig. 3.1. The call to* p *(resp.* q*) at L3 (resp. L4) creates an* active *task that becomes* finished *at L8 (resp. L6). In B, the call to*

r at L13 creates a pending *task that becomes* active *at L19 and* finished *after L19. p is an* active *task from L14 up to the end of the method. p will never become a* finished *task as its associated future variable is reused in L16 losing its association to p.*

The symbolic values used to describe the status of a task, referred to as MHP *atoms*, can be one of the following:

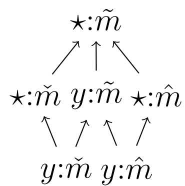
1. $y{:}\tilde{m}$, which represents an *active* task that is an instance of method m;
2. $y{:}\hat{m}$, which represents a *finished* task that is an instance of method m; and
3. $y{:}\check{m}$, which represents a *pending* task that is an instance of method m.

In the three cases the task is associated to the future variable y. In addition, since it is not always possible to relate tasks to future variables (e.g., if they are reused), we also allow symbolic values in which y is replaced by $\star$, i.e., $\star$ represents any future variable. $\star$ is also used when a call is made without storing the its return value.

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

Example 4.1.2. *Consider programs A, B and D. The multisets $\{y{:}\tilde{p}, z{:}\tilde{q}\}$, $\{y{:}\tilde{p}, z{:}\hat{q}\}$, $\{y{:}\hat{p}, z{:}\hat{q}\}$, $\{y{:}\check{r}, z{:}\tilde{p}\}$, $\{y{:}\check{r}, \star{:}\tilde{p}, \star{:}\tilde{p}, z{:}\hat{q}\}$ and $\{y{:}\tilde{p}, y{:}\tilde{q}\}$ respectively. describe the abstract states at L5, L7, L9, L15, L18 and L37. An important observation is that, in the multiset of L18, when the future variable is reused, its former association is lost (and hence becomes $\star$). However, multiple associations to one future variable can be kept when they correspond to disjunctive branches, as in L37.*

For a given program P, the set of all MHP atoms $\mathcal{A} = \{y{:}x \mid m \in P_{\mathcal{M}}, x \in \{\tilde{m}, \hat{m}, \check{m}\}, y \in P_{\mathcal{F}} \cup \{\star\}\}$ is a partially order set w.r.t. the partial order relation $\preceq$ defined as in the diagram below (we use $\prec$ for strict inequality and $=$ for syntactic equality).



The meaning of $a \preceq a'$ is that concrete scenarios described by a, are also described by a'. The bigger an atom is, the more general it is. In particular, the most general case is $\star{:}\tilde{m}$ that stands for an *active* method m, which can be running at any program point, and associated to any future variable. Any other atom of m is contained in it.

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

Figure 4.1: *Method-level MHP transfer function:* $\tau : s \times \mathcal{B} \mapsto \mathcal{B}$.

We have $y{:}\check{m} \preceq y{:}\tilde{m}$ and $y{:}\hat{m} \preceq y{:}\tilde{m}$ because $y{:}\check{m}$ and $y{:}\check{m}$ are included in the description of $y{:}\tilde{m}$ since an active task can be at any program point (including the entry program point and exit points). Besides, $y{:}x \preceq \star{:}x$ because any future variable is included in $\star$.

The set of all multisets over $\mathcal{A}$ is denoted by $\mathcal{B}$. We write $(a, i) \in M$ to indicate that a appears exactly $i > 0$ times in M. In the examples, we omit i when it is 1. Given $M_1, M_2 \in \mathcal{B}$, we say that $a \in M_2$ *covers* $a' \in M_1$ if $a' \preceq a$. Thus, $M_1 \sqsubseteq M_2$ if all elements of M_1 are covered by *different* elements from M_2.

Note that for two different $M_1, M_2 \in \mathcal{B}$, it might be the case that $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$, in such case they represent the same concrete states. This happens because when $(a, \infty) \in M$, then any $(a', i) \in M$ is redundant if $a' \preceq a$.

The join (or upper bound) of M_1 and M_2, denoted $M_1 \sqcup M_2$, is an operation that calculates a multiset $M_3 \in \mathcal{B}$ such that $M_1 \sqsubseteq M_3$ and $M_2 \sqsubseteq M_3$. It is not guaranteed that least upper bound exists, as we show in the following example.

Example 4.1.3. *Let* $M_1 = \{y{:}\hat{m}, y{:}\check{m}\}$ *and* $M_2 = \{y{:}\tilde{m}\}$. *Both* $M_3 = \{y{:}\hat{m}, y{:}\tilde{m}\}$ *and* $M_3' = \{y{:}\check{m}, y{:}\tilde{m}\}$ *are upper bounds for* M_1 *and* M_2. *However, there is no other upper bound* M_3'' *such that* $M_3'' \sqsubseteq M_3$ *and* $M_3'' \sqsubseteq M_3'$. *Thus, the least upper bound of* M_1 *and* M_2 *does not exist.*

The above example shows that there are several possible ways of computing an upper bound M_3 of two given abstract states M_1 and M_2. A possible instance of this algorithm is specified in Chap. 5.
In what follows, for the sake of simplicity, we assume that the program to be analyzed has been instrumented to have a **release** instruction before every **await** $y?$. This is required to simulate the auxiliary instruction **release** introduced in the semantics described in Chap. 2.1 (we could simulate it implicitly in the analysis also).

The analysis of a program P is done as follows. For each method $m \in P_{\mathcal{M}}$, it starts from an abstract state $\emptyset \in \mathcal{B}$, which assumes that there are no tasks executing (since we are looking at the locally invoked tasks), and propagates the information to the different

program points by applying the transfer function τ defined in Fig. 4.1 on the code *body*(m). The transfer function defines the effect of executing each (simple) instruction on a given abstract state $M \in \mathcal{B}$. Let us explain the different cases of τ:

- Case 1 adds an active instance of m to the abstract state;
- Case 2 adds a pending instance of m to the abstract state;
- Case 3 changes the status all active tasks that are guaranteed to be finished;
- Case 4 changes all pending tasks to active tasks; and
- Case 5 applies to the remaining instructions which do not have any effect on the MHP information.

Example 4.1.4. *Consider program B. The abstract state at L13 is $\emptyset$ since we have not invoked any method yet. Executing L13 adds $y{:}\check{r}$ since the call is to a method in the same object; executing L14 adds $z{:}\tilde{p}$; executing L16 renames one $z{:}\tilde{p}$ to $\star{:}\tilde{p}$ since the future variable z is reused, and adds $z{:}\tilde{q}$; executing L17 renames $z{:}\tilde{q}$ to $z{:}\hat{q}$ since it is guaranteed that q has finished. The auxiliary* ***release*** *between L18 and L19 renames $y{:}\check{r}$ to $y{:}\tilde{r}$, since the current task might suspend and thus any pending task might become active. Finally, L19 renames $y{:}\tilde{r}$ to $y{:}\hat{r}$.*

The analysis merges abstract states at branching points (i.e., after **if** and at loop entries) using the join operation $\sqcup$. The analysis of while loops requires iterating the corresponding code several times until a fixpoint is reached. To guarantee convergence in such cases we employ the following widening operator $\triangle : \mathcal{B} \times \mathcal{B} \mapsto \mathcal{B}$ after some predetermined number of iterations.

Briefly, assuming that M_2 is the current abstract state at the loop entry program point, and that $M_1 \sqsubseteq M_2$ is the abstract state at the previous iteration, then $M_1 \triangle M_2$ replaces each element $(a, i) \in M_2$ by (a, ∞) if $(a, j) \in M_1$ and $i > j$, i.e., it replaces unstable elements by infinite number of occurrences in order to stabilize them.

Example 4.1.5. *Let us demonstrate the analysis of the* ***if*** *and* ***while*** *statements on programs C and D. (**if**) At L37, the information that comes from the* ***then*** *and* ***else*** *branches is joined using $\sqcup$, namely $\{y{:}\tilde{p}\} \sqcup \{y{:}\tilde{q}\} = \{y{:}\tilde{p}, y{:}\tilde{q}\}$. Note that this state describes that either* q *or* p *are running at L37, but not both (as they share the same future variable); (**while**) In the first visit to L23, we have the abstract state $M_0 = \emptyset$, abstractly executing the body we reach L23 again with $M_1 = \{y{:}\hat{q}, z{:}\tilde{p}\}$ and joining it with M_0 results in M_1 itself. Similarly, if we apply two more iterations we respectively get $M_2 = \{\star{:}\hat{q}, \star{:}\tilde{p}, y{:}\hat{q}, z{:}\tilde{p}\}$ and*

$M_3 = \{(\star{:}\hat{q}, 2), (\star{:}\tilde{p}, 2), y{:}\hat{q}, z{:}\tilde{p}\}$. Inspecting M_2 and M_3, we see that $\star{:}\hat{q}$ and $\star{:}\tilde{p}$ are unstable, thus, we apply the widening operator $M_2 \triangle M_3$ obtaining $M_3' = \{(\star{:}\hat{q}, \infty), (\star{:}\tilde{p}, \infty), y{:}\hat{q}, z{:}\tilde{p}\}$. Executing the loop body starting with the new abstract state does not add any new MHP atoms since $\star{:}\hat{q}$ and $\star{:}\hat{p}$ already appear an infinite number of times.

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

Example 4.1.6. *The following table summarizes $\mathcal{L}_P$ for some selected program points of interest (from Fig. 3.1) that we will use in the next section:*

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

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

4.2 The Notion of MHP Graph

We now introduce the notion of *MHP graph* from which it is possible to extract precise information on which program points might globally run in parallel (according to Def. 3.0.2). A MHP graph has different types of nodes and different types of edges. There are nodes that represent the status of methods (active, pending or finished) and nodes which represent the program points. Outgoing edges from method nodes represent points of which at most one might be executing. In contrast, outgoing edges from program point nodes represent tasks such that any of them might be running. The information computed by the method-level MHP analysis is required to construct the MHP graph. When two nodes are directly connected by $i > 0$ edges, we connect them with a single edge of weight i. We start by formally constructing the MHP graph for a given program P, and then explain the construction in detail.

Definition 4.2.1 (MHP graph)**.** *Given a program P, and its method-level MHP analysis result $\mathcal{L}_P$, the MHP graph of P is a directed graph $\mathcal{G}_P = \langle V, E\rangle$ with a set of nodes V and a set of edges $E = E_1 \cup E_2 \cup E_3$ defined as follows:*

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

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

1. *Method nodes*: Each $m \in P_{\mathcal{M}}$ contributes three nodes $\tilde{m}$, $\hat{m}$, and $\check{m}$. These nodes will be used to describe the program points that can be reached from active, finished or pending tasks which are instances of m.

2. *Program point nodes*: Each $p \in P_{\mathcal{P}}$ contributes a node p that will be used to describe which other program points might be running in parallel with it.

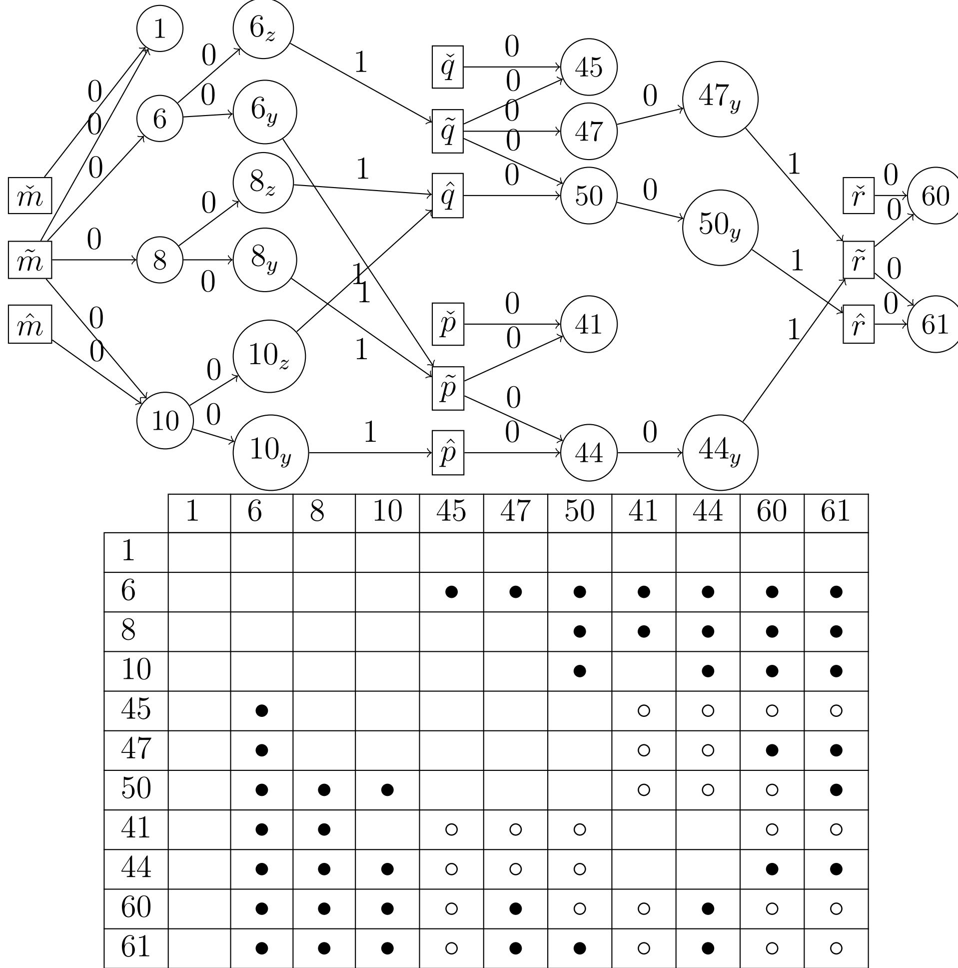
3. *Future variable nodes*: These nodes are a refinement of program point nodes for improving precision in the presence of branching constructs. Each future variable y that appears in $\mathcal{L}_P(p)$ contributes a node p_y. These nodes will be used to state that if there are several MHP atoms in $\mathcal{L}_P(p)$ that are associated to y, then at most one of them can be running.

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

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

2. Edges in E_2 describe which tasks might run in parallel with such program point. For every program point $p \in P_{\mathcal{P}}$, if $(\star{:}x, i) \in \mathcal{L}_P(p)$ then $p \xrightarrow{i} x$ is added to E_2. This edges means, if $x = \tilde{m}$ for example, that up to i instances of m might be running in parallel when reaching p;

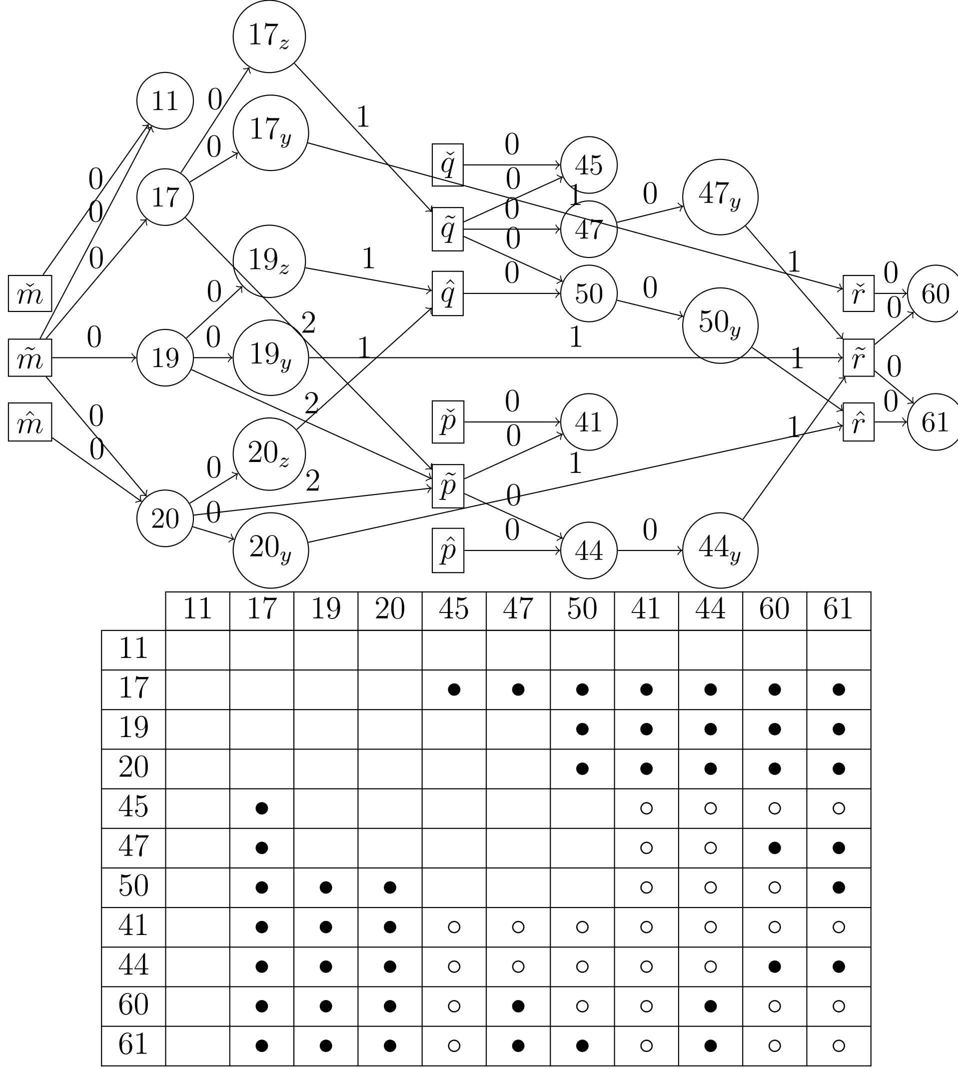
3. Edges in E_3 enrich the information for each program point given in E_2. An edge $p_y \xrightarrow{1} x$ is added to E_3 if $(y{:}x, i) \in \mathcal{L}_P(p)$. For each future variable y that appears in $\mathcal{L}_P(p)$ an edge $p \xrightarrow{0} p_y$ is also added to E_3. This allows us to accurately handle cases in which several MHP atoms in $\mathcal{L}_P(p)$ are associated to the same future variable. Recall that in such cases at most one of the corresponding tasks can be available (see Ex. 4.1.2).



	1	6	8	10	45	47	50	41	44	60	61
1											
6					●	●	●	●	●	●	●
8							●	●	●	●	●
10							●		●	●	●
45		●						○	○	○	○
47		●						○	○	●	●
50		●	●	●				○	○	○	●
41		●	●		○	○	○			○	○
44		●	●	●	○	○	○			●	●
60		●	●	●	○	●	○	○	●	○	○
61		●	●	●	○	●	●	○	●	○	○

Figure 4.2: *The $\mathcal{G}_P$ of example A (up) and its corresponding $\tilde{\mathcal{E}}_P$ (down).*

Example 4.2.2. *Using the method-level MHP information of Ex. 4.1.6 we obtain the MHP graphs for the four examples. These graphs are displayed on the left side of Fig. 4.2, Fig. 5.2,Fig. 4.4 and Fig. 4.5. For readability, the graphs do not include all program points, but rather only those that correspond to entry, get and release points. However, not including all program points does not harm the results validity (See Sec. 4.4).*



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

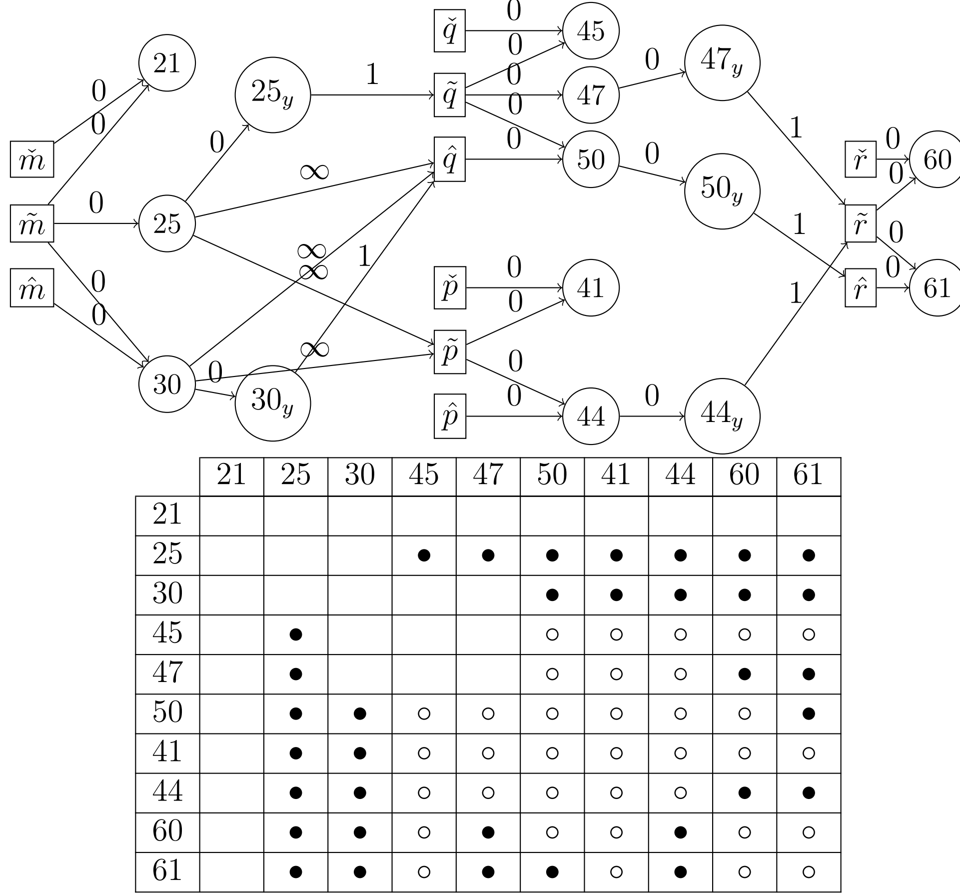
Figure 4.3: *The $\mathcal{G}_{\mathcal{P}}$ of example B (up) and its corresponding $\tilde{\mathcal{E}}_P$ (down).*

4.3 Inference of Global MHP

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

1. there is a non-empty path in $\mathcal{G}_{\mathcal{P}}$ from p_1 to p_2 or vice-versa; and

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

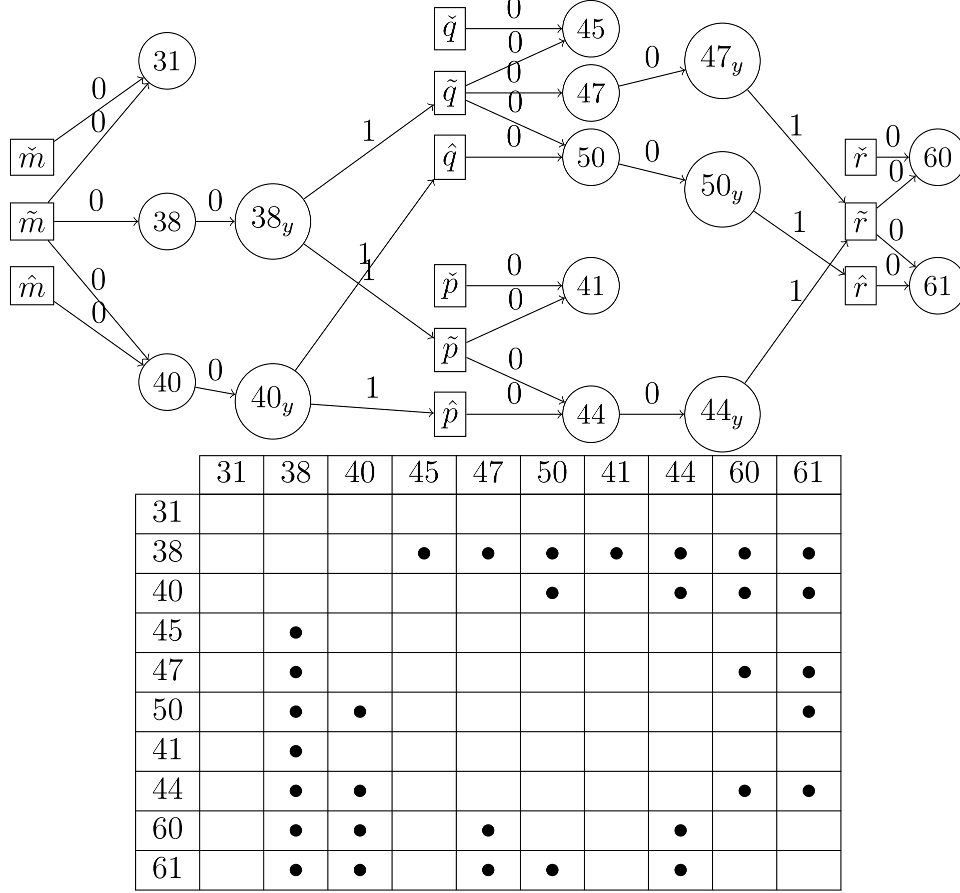


	21	25	30	45	47	50	41	44	60	61
21										
25				●	●	●	●	●	●	●
30						●	●	●	●	●
45		●				○	○	○	○	○
47		●				○	○	○	●	●
50		●	●	○	○	○	○	○	○	●
41		●	●	○	○	○	○	○	○	○
44		●	●	○	○	○	○	○	●	●
60		●	●	○	●	○	○	●	○	○
61		●	●	○	●	●	○	●	○	○

Figure 4.4: *The $\mathcal{G}_P$ of example C (up) and its corresponding $\tilde{\mathcal{E}}_P$ (down).*

The first case corresponds to *direct MHP* scenarios in which, when a task is running at p_1, there is another task running from which it is possible to *transitively* reach p_2, or vice-versa. This is the case, for example, of program points 17 and 50 in Fig. 5.2.

The second case corresponds to *indirect MHP* scenarios in which a task is running at p_3 and there are two other tasks p_1 and p_2 executing in parallel and both are reachable from p_3. This is the case, for example, of program points 50 and 44 that are both reachable from program point 19 through paths that start with a different edge in the example B (Fig. 5.2). Observe that the first edge can only be shared if it has weight $i > 1$ because it represents that there might be more than one instance of the same type of task running. This allows us to infer that 41 may run in parallel with itself because the edge from 17 to $\tilde{p}$ has weight 2 and, besides, that 41 can run in parallel with 44. Note that program points 45, 47, and 50 of method q do not satisfy any of the above conditions, which implies, as expected, that they cannot run in parallel.



	31	38	40	45	47	50	41	44	60	61
31										
38				•	•	•	•	•	•	•
40						•		•	•	•
45		•								
47		•							•	•
50		•	•							•
41		•								
44		•	•						•	•
60		•	•		•			•		
61		•	•		•	•		•		

Figure 4.5: *The $\mathcal{G}_{\mathcal{P}}$ of example D (up) and its corresponding $\tilde{\mathcal{E}}_P$ (down).*

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

Definition 4.3.1. *Given a program P, we let $\tilde{\mathcal{E}}_P = \mathit{directMHP} \cup \mathit{indirectMHP}$ where*

$$\begin{array}{rl} \mathit{directMHP} = & \{(p_1,p_2) \mid \; p_1, p_2 \in P_{\mathcal{P}}, p_1 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}})\} \\ \mathit{indirectMHP} = & \{(p_1,p_2) \mid \; p_1, p_2, p_3 \in P_{\mathcal{P}}, p_3 \xrightarrow{i} x_1 \rightsquigarrow p_1 \in \mathcal{G}_{\mathcal{P}}, p_3 \xrightarrow{j} x_2 \rightsquigarrow p_2 \in \mathcal{G}_{\mathcal{P}}, \\ & \quad x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1)\} \end{array}$$

Example 4.3.2. *The tables on the right side of Fig. 4.2, Fig. 5.2 Fig. 4.4 and Fig. 4.5 represent the $\tilde{\mathcal{E}}_P$ obtained from the graph on the left side. Empty cells mean that the corresponding points cannot run in parallel. Cells marked by • indicate that the pair is in directMHP. Cells marked with ∘ indicate that the pair is in indirectMHP. Note that the tables capture the MHP relations discussed in Sec. 3.1. In fact, the analysis achieves full precision for the considered points in all the examples.*

4.4 Partial analyses and points of interest

As mentioned before, many program points can be safely ignored if we are not interested in their parallelism information. In this section, we define the notion of points of interest, partial MHP analysis and a sufficient condition for a program point to be safely ignored. This condition does not take into account the instruction at that program point, but rather defined in terms of the result of method-level analysis. Then, we reexamine this condition from the perspective of the instruction being executed at that point, and extract those instruction that are essential, i.e., for which the corresponding program points should be included in the analysis.

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

Definition 4.4.1 (Partial MHP information)**.** *The partial MHP information of P w.r.t $iP_{\mathcal{P}}$ is defined as $p\tilde{\mathcal{E}}_P = \tilde{\mathcal{E}}_P \cap (iP_{\mathcal{P}} \times iP_{\mathcal{P}})$.*

The partial MHP information consists of those pairs that involve program points from $iP_{\mathcal{P}}$. Our interest is to compute $p\tilde{\mathcal{E}}_P$ directly, and not by computing $\tilde{\mathcal{E}}_P$ and then restricting it to $p\tilde{\mathcal{E}}_P$ as in the above definition. In what follows we describe a partial MHP analysis for inferring $\tilde{\mathcal{E}}_P$.

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

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

Corollary 4.4.2. *Let $p \in m$ be a program point such that $p \notin iP_{\mathcal{P}}$. If $p \neq p_{\mathring{m}}$,$p \neq p_{\dot{m}}$ and $\exists p' \in m$ such that $\mathcal{L}_{P}(p) \preceq \mathcal{L}_{P}(p')$, then p can be safely ignored w.r.t. $iP_{\mathcal{P}}$.*

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

Let us see how in practice we identify a program point that satisfy Cor. 4.4.2, depending on the instruction being executed at that point. Let s be a sequence of statements in a method m and $pp(s) = p$. If $s = instr; s'$ and $pp(s') = p'$, by the definition of the transfer

function τ, we have $\mathcal{L}_{P}(p) \preceq \mathcal{L}_{P}(p')$ for all *instr* that are not **await** y? or y.**get**. That is, $\mathcal{L}_{P}$ always grows except for **await** y? and y.**get** instructions. If we apply Cor. 4.4.2, we see that all points except $p_{\mathring{m}}$, $p_{\dot{m}}$, **await** y? and y.**get** can be ignored.

Furthermore, we know that $\mathcal{L}_{P}(p_{\mathring{m}}) = \emptyset$ for all methods, which implies that entry program points do not have outgoing edges in $\mathcal{G}_{P}$. Consequently, if $p_{\mathring{m}} \notin iP_{\mathcal{P}}$, $p_{\mathring{m}}$ cannot be part of any path between program points in $iP_{\mathcal{P}}$, nor be a common ancestor of any program points. Therefore, $p_{\mathring{m}}$ can also be safely ignored.

4.5 Soundness and Complexity

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

Theorem 4.5.1 (soundness)**.** $\mathcal{E}_P \subseteq \tilde{\mathcal{E}}_P$.

As regards complexity, we distinguish its three phases.

1. The $\mathcal{L}_{P}$ computation can be performed independently for each method m. The transfer function τ only needs to be applied a constant number of times for each program point, even for loops, due to the use of widening. It is possible to represent multisets such that the cost of all multiset operations is linear w.r.t. their sizes which are at most $nm_m \cdot fut_m$ (see Appendix 5.2). Where nm_m is the number of different methods that can be called from m and fut_m is the number of future variables in m. Therefore, the cost of computing $\mathcal{L}_{P}$ for a method m is in $O(pp_m \cdot nm_m \cdot fut_m)$ where pp_m is the number of program points in the method.

2. The cost of creating the graph $\mathcal{G}_{P}$ is linear with respect the number of edges. The number of edges originated from a method m is in $O(pp'_m \cdot nm_m \cdot fut_m)$ where pp'_m is the number of program points of interest.

   A strong feature of our analysis is that most of program points can be ignored in this phase without affecting correctness or precision. Only points that correspond to **await** and **get** instructions and exit points are required for correctness (see Sec 4.4).

   For instance, if we were interested is in observing how tasks can interleave their executions, the points of interests are those at which the processor switches among tasks, namely the **await** and **get** instructions and at the entry points of methods. Thus, when constructing the graphs we need to include the required points plus the points of interest (as in the examples of Fig. 4.2, Fig. 5.2 Fig. 4.4 and Fig. 4.5).

3. Once the graph has been created, computing $\tilde{\mathcal{E}}_P$ is basically a graph reachability problem. Therefore, a straightforward algorithm for inferring of $\tilde{\mathcal{E}}_P$ is clearly in $O(n^3)$ where n is the number of nodes of the graph. However, a major advantage of this analysis, is that for most applications there is no need to compute the complete $\tilde{\mathcal{E}}_P$, but rather obtain this information *on demand*.

Chapter 5

MayPar Tool: Implementation and Experimental Evaluation

MayPar is a May-happen-in-PARallel analyzer for ABS [3], a distributed asynchronous language based on *concurrent objects* [?]. The tool implements the MHP analysis described in the previous sections. The information gathered by the analysis provides a global perspective of the communication and synchronization among objects and enables a better comprehension of the tasks interleavings that might occur along the program execution.

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. MayPar can be used (1) to spot bugs in the program related to fragments of code which should not run in parallel and also (2) to improve the precision of other analyses which infer more complex properties (e.g., termination and resource consumption).

The web interface of MayPar can be tried out online at: `http://costa.ls.fi.upm.es/costabs/mhp`.

5.1 Implementation Details

This section reports on some aspects of the real implementation that were not completely specified in the general description of the analysis such as the Method domain operations and optimizations that differ from the original description.

5.2 Method-level analysis state representation and operations

One of the aspects that makes the method-level analysis efficient is the low cost of the operations over the multiset domain. In what follows, the specific representation of the method-level states is described and how the most important operations are performed (namely State Inclusion and Upper Bound).

5.2.1 State Representation

Given a multiset M, all atoms that refer to the same method m can be represented with a tuple $\langle \mathbb{P}_m, \mathbb{A}_m, \mathbb{E}_m, p_m, a_m, e_m \rangle$ where:

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

  Assuming future variables are enumerated $y_1, y_2 ... y_n$: $\mathbb{P}[i] = 1 \Leftrightarrow y_i{:}\check{m} \in M$; $\mathbb{A}[i] = 1 \Leftrightarrow y_i{:}\tilde{m} \in M$; and $\mathbb{E}[i] = 1 \Leftrightarrow y_i{:}\hat{m} \in M$.

- p_m a_m e_m are natural numbers or infinite. Their domain is $dom(p) = dom(a) = dom(e) = \mathbb{N}^*$ where $\mathbb{N}^* = \mathbb{N} \cup \{\infty\}$.

  Their values represent the following: $p = i \Leftrightarrow (\star{:}\check{m}, i) \in M$, $a = i \Leftrightarrow (\star{:}\tilde{m}, i) \in M$, and $e = i \Leftrightarrow (\star{:}\hat{m}, i) \in M$.

Therefore, a multiset M that results from the analysis of a method m' can be represented as a list of tuples $\langle \mathbb{P}_m, \mathbb{A}_m, \mathbb{E}_m, p_m, a_m, e_m \rangle$ where m is any method that can be called from m'. Note that the maximum length of the list is nm'_m (the number of different methods that can be called from m').

5.2.2 Inclusion operation

In order to see if $M_1 \preceq M_2$, we consider each tuple $\langle \mathbb{P}_m, \mathbb{A}_m, \mathbb{E}_m, p_m, a_m, e_m \rangle$ separately because there cannot be any relation between elements of different tuples (as they refer to different methods). For each tuple t_1 in M_1 that corresponds to a method m, we need to check that its corresponding one in M_2, t_2, can cover all its elements.

For that purpose we use the concept of safe transformation.

Definition 5.2.1. *A* safe transformation *is one in which we only loose information. i.e., transforming M into M' is safe as long as $M \preceq M'$. A* basic safe transformation *consists in substituting y for x such that $x \preceq y$.*

If $M_1 \preceq M_2$, we should be able to go from M_1 to M_2 through a sequence of basic safe transformations. Given our state representation, we can easily check if such transformation is possible. For each method m:

- We start by subtracting t_1 from t_2. We obtain $t' = \langle \mathbb{P}'_m, \mathbb{A}'_m, \mathbb{E}'_m, p'_m, a'_m, e'_m \rangle$ $t' = t'_2 - t_1$. Note that elements in $\mathbb{P}', \mathbb{A}'$ and $\mathbb{E}'$ are in $B' = \{0, 1, -1\}$ and p', a' and e' can take values in $\mathbb{Z}^* = \mathbb{Z} \cup \{\infty, -\infty\}$. In this representation, a negative number represents an MHP atom of M_1 that has not been covered by one in M_2. On the other hand, a positive number represents a MHP atom of M_2 that has not covered any atom yet. Now, a *basic safe transformation* consists in taking negative numbers from one category and adding them to a higher category (according to the partial order defined in Sec. 4.1). However, we have to perform these transformations in a specific order:

  1. If $a' < 0$, we finish with $M_1 \not\preceq M_2$. Those elements cannot be covered (as they correspond to $\star{:}\tilde{m}$).
  2. Elements in $\mathbb{A}'$, p' and e' can only be covered by elements in a'. We move any negative number from $\mathbb{A}'$, p' and e' to a' as long as a' stays non-negative. If we cannot remove all negative numbers in those three elements, we conclude that $M_1 \not\preceq M_2$.
  3. Then we have to cover the elements of $\mathbb{P}'$ and $\mathbb{E}'$. Those elements can be covered by elements in $\mathbb{A}'$, p', e' and a'. However, each positive element in $\mathbb{A}'$ can only cover elements with the same future variable. That is, $\mathbb{A}'[i]$ can only cover $\mathbb{P}'[i]$ or $\mathbb{E}'[i]$ whereas elements in p' (resp e') and a' and can cover any element of $\mathbb{P}'$ (resp $\mathbb{E}'$).
     - For each positive element $\mathbb{A}'[i]$, if it can only cover one element in $\mathbb{P}'$ or $\mathbb{E}'$ (for example, $\mathbb{P}'[i] \geq 0$ and $\mathbb{E}'[i] < 0$), we apply the corresponding transformation ($\mathbb{E}'[i] = 0; \mathbb{A}'[i] = 0$; in the previous example).
     - Afterwards, we count the elements of $\mathbb{A}'$ that can cover two elements one from $\mathbb{P}'$ and one from $\mathbb{E}'$. We call it bc.
     - We count the negative elements in $\mathbb{P}'$ (np) and $\mathbb{E}'$ (ne). $rp = (p' - np)$ if $np > p'$, $rp = 0$ otherwise. That is, rp encodes (with a negative number) the atoms that cannot be covered by p'. Likewise, $re = (e' - ne)$ if $ne > e'$, $re = 0$ otherwise. if $(rp + re) + (bc + a)' < 0$, $M_1 \not\preceq M_2$. Otherwise, we continue with the next tuple of M_1 and M_2.

If after examining all tuples in M_1 we do not obtain a negative answer, we can safely conclude that $M_1 \preceq M_2$.

5.2.3 Upper bound operation

We want to compute a M_3 such that $M_1 \preceq M_3$ and $M_2 \preceq M_3$. In general there is not least upper bound as stated in Sec. 4.1. For each method m that can be called from the analyzed method m', we examine the corresponding tuples t_1 and t_2. If t_1 is not present we just keep t_2. If both are present, we need to compute a t' such that it covers both t_1 and t_2.

If $t_1 \preceq t_2$ or $t_2 \preceq t_1$ we have that $t' = t_2$ or $t' = t_1$ respectively. Otherwise, we need to define an operation that for a t_1 and t_2 obtains t' such that $t_1 \sqsubseteq t'$ and $t_2 \sqsubseteq t'$.

For each $t_1 \in M_1$, $t_2 \in M_2$ that correspond to a method m, let $t_1 = \langle \mathbb{P}_1, \mathbb{A}_1, \mathbb{E}_1, p_1, a_1, e_1 \rangle$ and $t_2 = \langle \mathbb{P}_2, \mathbb{A}_2, \mathbb{E}_2, p_2, a_2, e_2 \rangle$ we build $t' = \langle \mathbb{P}', \mathbb{A}', \mathbb{E}', p', a', e' \rangle$ as follows:

1. We put any infinite that appears in t_1 or t_2 in t'.
2. Any element in t_1 or t_2 that can be covered by an infinite in t' is removed.
3. The common elements in t_1 and t_2 are removed and added to t'.
4. We make safe tranformations to increment the number of matches (the number of common elements). This is because, in general, we prefer having the smallest possible number of atoms.

   The (heuristic) order in which these safe tranformations are performed is the following:

From $\mathbb{P}$ to $\mathbb{A}$ If $\mathbb{P}_1[i] = 1$ and $\mathbb{A}_2[i] = 1$, $\mathbb{A}'[i] := 1$, $\mathbb{P}_1[i] := 0$ and $\mathbb{A}_2[i] := 0$. If $\mathbb{P}_2[i] = 1$ and $\mathbb{A}_1[i] = 1$, $\mathbb{A}'[i] := 1$, $\mathbb{P}_1[i] := 0$ and $\mathbb{A}'_2[i] := 0$. Note that both possibilities cannot happen for the same index i because common elements have been already removed.

From $\mathbb{P}$ to p As only one of p_1 and p_2 can be positive (we removed common elements), the promotion can only be done in one sense. If p_1 is positive (resp p_2 positive), we remove elements from $\mathbb{P}_2$ (resp $\mathbb{P}_1$), decrement p_1 (resp p_2) and increment p' as long as $p_1 > 0$ (resp $p_2 > 0$).

From $\mathbb{P}$ to a It is performed as in 4.

From p to a Only one of p_1 and p_2 and one of a_1 and a_2 can be positive. If p_1 and a_2 are positive or p_2 and a_1 are positive, we take the minimum of the two positive elements and add it to a'. That minimum is subtracted from the two elements in t_1 and t_2.

From $\mathbb{E}$ to $\mathbb{A}$ It is performed as in 4.

From $\mathbb{E}$ to e It is performed as in 4.

From $\mathbb{E}$ to a It is performed as in 4.

From e to a It is performed as in 4.

5. Finally, the remaining elements in t_1 and t_2 are the ones that could not be matched. These are simply added to t'.

5.2.4 Optimizations

In order to obtain the best performance, it is important to reduce the size of the graph as much as possible without losing either precision or soundness. There are two optimizations that have been implemented in MayPar:

1. In most occasions a future variable y is associated to a single method m. In such cases, we have a future variable node pp_y that has only one incoming edge from pp and one outgoing edge to the corresponding method node of m. It is clear that the node pp_y can be removed and substituted by a direct edge from the node pp to the method node reducing the size of the graph.

2. When creating the graph, *pending* method nodes can be safely ignored for some uses of the analysis. In particular, if we are only interested in pairs of program points that can progress at the same time.

   pending method nodes only create paths to entry proints. However, entry points never have outgoing edges (their method-level state is $\emptyset$). They cannot be intermediate nodes of other paths. Consequently, by ignoring *pending* method nodes, we only lose pairs of the form $(p_{\mathring{m}}, pp)$ where m is *pending*. But we know that a *pending* method cannot be executing. Therefore, the pairs we lose are not necessary (and could even be misleading) in this kind of scenario.

   Note that *pending* methods are still used in method-level analysis and they are only ignored when constructing the MHP graph.

5.3 Use of the Tool

We illustrate the use of MayPar through an example showing how it can aid the programmer in debugging and understanding the concurrent behavior of the program and how it can aid other program analyzers in sounly approximating the program's global state at context switching points.

```
class User(String email) {
  List<String> msgs;
  User receive(String m) {
    msgs = Cons(m,msgs);
  }
}

class AddrBook(List<User> users) {
  User getUser(String email) {
     ...
  }
}

class Notifier(AddrBook ab) {
  List<String> addrs = Nil;
  Unit notify(String m) {
    while ( addrs != Nil ) {
      Fut<User> u;
      u = ab!getUser(head(addrs));
      await u?;
      User us = u.get;
      us!receive(m);
      addrs = tail(addrs);
    }
  }

  Unit addAddr(String u) {
    addrs = Cons(u, addrs);
  }

  Unit addAddrs(List<String> l) {
    while ( l != Nil ) {
     addAddr(head(l));
     l=tail(l);
    }
  }
}

main{
  User u1 = new User("a@b.com");
  ...
  AddrBook ab = new AddrBook([u1,...]);
  Notifier ms = new Notifier(ab);
  Fut<Unit> x = ms!addAddrs(["a@b.com",...
  await x?
  ms.notify("Hello ...");
}
```

Figure 5.1: *The* Notifier *example*

5.3.1 The Notifier Example

We use the program depicted in Fig. 5.1 which implements several classes to model users in a distributed environment, and the processes of notifying them with messages.

Class User (L1-6) models a user which has a field email (declared as a class parameter) for storing its associated email address, and a field msgs for storing the received messages. Method receive is used to send a message to the user.

Class AddrBook (L8-12) models an address book which has a field users that contains a list of registered users, and a method getUser for retrieving the object (of type User) that corresponds to a given email address. The code of this method is omitted, we just assume that it is completely sequential, and does not call any other method.

Class Notifier (L14-37) models the process of notifying users with messages. Field ab contains an AddrBook object which is used to retrieve users by email addresses. Field addrs

contains a list of registered email addresses to be notified. Method addAddrs adds a given list of email addresses to field addrs, by asynchronously calling addAddr on each of them. Method notify is used to notify all registered users with a given message m. It iterates over the list addrs, and at each iteration:

1. it requests, by calling getUser at L19, the User object that corresponds to the first email address in list addrs, and, at L21, waits until it gets the result back. Note that the call to getUser is asynchronous, and that the **await** instruction blocks the execution of notify, allowing other pending methods to execute in the meanwhile. The instruction **get** is used to retrieve the result of the asynchronous call;

2. it sends the message m to the corresponding user by asynchronously calling the corresponding receive method at L22; and

3. it removes the first email address from addrs at L23.

Method main implements the following usage scenario:

1. it creates several User objects at L40-41, each with a unique email address;

2. it creates an AddrBook object at L42, and passes it a list of users [u1 ,...] ;

3. it creates a Notifier object at L43 which receives the address book ab as a parameter;

4. it adds some email addresses to be notified by asynchrounsouly calling addAddrs at L44, and waits at L45 until it has terminated; and

5. finally, it calls method notify at L46 in order to notify all registered users with a given message.

5.3.2 MHP Analysis of the Notifier example

In a first step, MayPar generates an MHP graph that captures all MHP relations between the different program points of the program. Then, using this graph, it outputs a set of MHP pairs of the form (i, j) which indicates that the instruction at program point i might execute in parallel with the one at program point j, and vice versa. This set can be obtained for all program points, for some program points of interest, or even on demand, e.g., querying if two program points might run in parallel. Although the MHP graph is shown in the output as a `.dot` file, the user is not required to understand its details and can simply ignore it. However, as we see in the next section, it might help in identifying the source of unexpected MHP pairs.

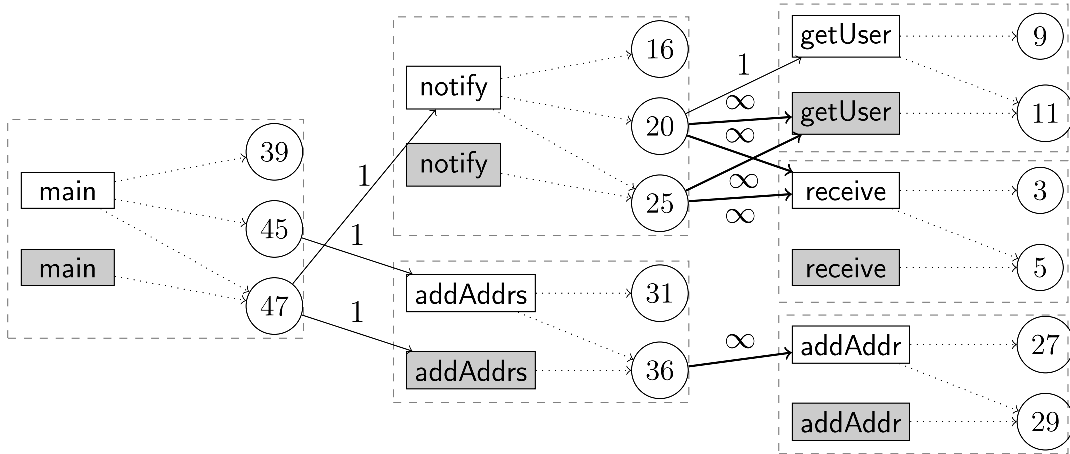


Figure 5.2: *MHP graph for the buggy program of Fig. 5.1*

Let us demonstrate the output of MAYPAR on the program of Fig. 5.1. The corresponding MHP graph is depicted in Fig. 5.2 (it contains the optimizations described in Section 5.2.4). Each program point i that corresponds to a context switch, i.e., a program point in which the execution might switch from one method to another, is represented by a node (i). These nodes always include the method's *entry* and *exit* program points, as it is required for soundness as mentioned before. Each method m contributes two nodes: [m] represents an instance of m that is *active*, i.e., running and can be at any program point, and [m] represents an instance of m that is *finished*, i.e., it is at the exit program point.

The MHP graph is composed from 5 subgraphs, one for each method. The nodes of each subgraph are grouped within a dashed rectangle. In each subgraph: (a) the *active* method node (the white rectangle) is connected to all program point nodes of that method, meaning that when the method is active it can be executing at any of those program points; and (b) the *finished* method node (the gray rectangle) is connected to the exit program point node, meaning that when the method is finished it must be at the exit program point. For example, in the subgraph of method main, there are edges from [main] to nodes (39), (45), and (47); and from [main] to (47).

The subgraphs are connected among them by weighted edges. Each such edge starts at a program point node in one subgraph, and ends in an active or finished method node in another subgraph (it can be the same if the method is recursive). As previously explained, these edges are inferred by applying the *method-level* MHP analysis.

For example, the method-level analysis infers:

- For method main, at L45, there might be one active instance of method addAddrs. This will add an edge from (45) to [addAddrs]. The edge is labeled with 1 to indicate that it

is only one instance of addAddrs; and

- for method notify, at L20, there might be an active instance of getUser, many finished instances of getUser, and many active instances of receive. This will add an edge from (20) to [getUser] with label 1, to [getUser] with label ∞, and to [receive] with label ∞.

Afterwards, our MHP property allow us to generate all the MHP pairs from the graph, or to provide them on demand. The set of MHP pairs is given in a simple text format, or as an XML structure to facilitate parsing when integrated within other tools.

We can see that there is a path from (45) to (27) which induce the direct MHP pair (45,27). Also, there are different paths from (47) to both (27) and (20) which induces the indirect MHP pair (20,27).

The web-interface provides an interactive way of showing these pairs: when clicking on a specific program point in the program, all program points that might run in parallel become highlighted.

5.3.3 Using MHP for debugging and understanding concurrent programs

While testing the program of Fig. 5.1, the programmer noticed that it does not have the expected behavior. In particular, it does not notify some users, and some others are notified several times. As we have seen above, MayPar reports the MHP pair (20,27). This means that while waiting for getUser to terminate at L20, an instance of addAddr might be executing and thus modifying field addrs.

This valuable information provides a hint that allows constructing the following unexpected scenario. Suppose that when entering the loop, field addrs equals to ["a@b.com"," b@c.com"]. Then, while waiting for the answer from getUser at L20, there is an instance of addAddr that executes in parallel and adds "c@d.com" to addrs. Thus, when reaching L23, addrs will be equal to ["c@d.com","a@b.com","b@c.com"], and removing the first element of this list means that "c@d.com" will not be notified, and that in the next iteration "a@b.com" will be notified again.

To understand the source of this error, the programmer inspects how the MHP information was obtained using the MHP graph of Fig. 5.2. First, the direct MHP relations are inspected for L20, by querying MayPar with this selected point. However, they do not lead to any error since (27) is not reachable from (20), and vice versa. Then, inspecting the indirect MHP relation, the programmer observes that L45 is the source of this error since (45) reaches both (20) and (27) on two different paths.

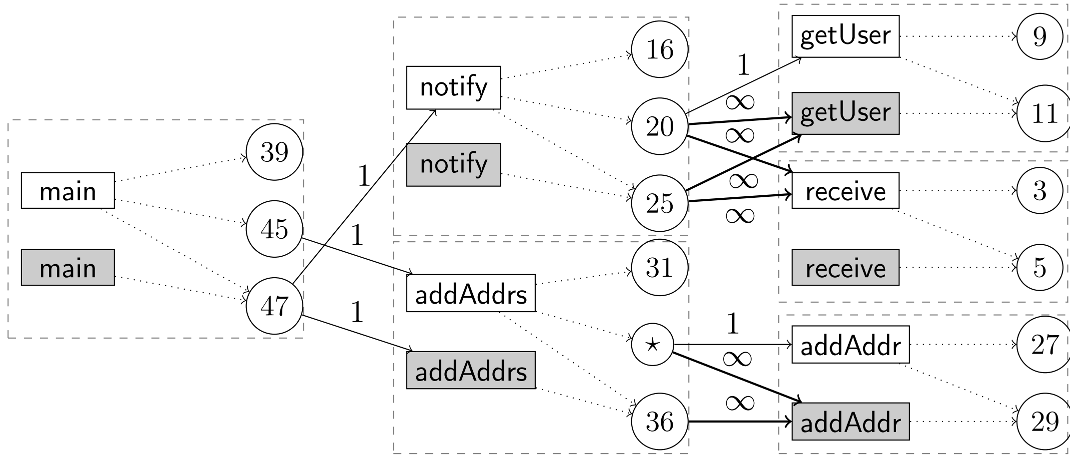


Figure 5.3: *MHP graph after correction of method* addAddrs

Tracking the MHP information back, the error can be easily identified: at L33, the call to addAddr is invoked asynchronously but it does wait for it to terminate, thus, it is scheduled and might execute later while notify is waiting at L20.

Adding an **await** instruction for the call immediately after L33 solves the problem. Indeed, applying MHP analysis on the new version provides the MHP graph in Fig. 5.3, where (⋆) corresponds to the new program point. Now we can see that the indirect MHP that involve (20) and (27) has been eliminated.

5.3.4 Using MHP information in static program analysis

Any static program analysis that aims at soundly reasoning on the behaviour of concurrent programs must take into account all possible interleavings between methods during the execution. In our language, it is clear where such interleavings might happen, e.g., *when* executing an **await** instruction. The MHP analysis can be used to complement this with information on *who* might interleave at these points.

For example, when automatically reasoning on the resource consumption or termination of the program of Fig. 5.1, the analysis must bound the number of iterations that each loop can make. For the **while** loop in method notify, such analysis is able to infer that its number of iterations is as the length of list addrs, under the condition that at L20, when the method suspends, the length of the list is not increased.

Using the MHP information from the MHP graph of Fig. 5.2, the analysis can verify that this condition might be violated since L27 might be executing in parallel. However, using the MHP graph of Fig 5.3 it can verify that it holds, since the only methods that can

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

Table 5.1: *Statistics about the analysis execution (times are in milliseconds)*

run in parallel are receive and getUser that do not call back addAddr.

5.4 Experimental Evaluation

Experimental evaluation has been carried out using two industrial case studies: ReplicationSystem and TradingSystem, which can be found at http://www.hats-project.eu, as well as few typical concurrent applications: PeerToPeer, a peer to peer protocol implementation; Chat, a client-server implementation of a chat program; MailServer, a simple model of a Mail server; BookShop, a web shop client-server application; BBuffer, a classical bounded-buffer for communicating several producers and consumers; and DistHT, a distributed hash-table.

Table 5.1 summarizes our experiments. They have been performed on an Intel Core i5 at 2.4GHz with 3.7GB of RAM, running Linux. For each program, $\mathcal{G}_P$ is built using only the program points required for soundness and the entry points. These graphs are useful to infer how the different tasks interleave. The relation $\tilde{\mathcal{E}}_P$ is completely computed for the given graphs. However, in these experiments, the MHP pairs that contain exit points have been ignored. Having a pair $(p, p_{\dot{m}})$, only implies that p can happen after m has finished executing but it does not give a good measure of the program parallelism.

- **Ns** is the number of nodes of $\mathcal{G}_P$.
- $\mathbf{NP}_{\mathcal{P}}$ is the number of program point nodes.
- $\mathbf{E_p}$ is the number of MHP pairs obtained by running the program using a random scheduler, i.e., one which randomly chooses the next task to execute when the processor is released. These executions are bounded to a maximum number of interleavings as termination in some examples is not guaranteed. Observe that $\mathbf{E_p}$ does not capture

all possible MHP pairs but just gives us an idea of the level of real parallelism. It gives us a lower bound of $\mathcal{E}_P$ which we will use to approximate the error.

- $\tilde{\mathcal{E}}_{\mathbf{P}}$ is the number of pairs inferred by the analysis.

- $\mathbf{PPs^2}$ is the square of the number of program points, i.e., the number of pairs considered in the analysis. $\mathbf{PPs^2} - \tilde{\mathcal{E}}_{\mathbf{P}}$ gives us the number of pairs that are guaranteed not to happen in parallel.

- $\mathbf{R}\varepsilon = 100(\tilde{\mathcal{E}}_{\mathbf{P}} - \mathbf{E_p})/\mathbf{PPs^2}$ is the approximated error percentage taking $\mathbf{E_p}$ as reference, i.e., $\mathbf{R}\varepsilon$ is an upper bound of the real error of the analysis.

- $\mathbf{T_G}$ is the time (in milliseconds) taken by the method-level analysis and in the graph construction.

- $\mathbf{T}_{\tilde{\mathcal{E}}_{\mathbf{P}}}$ is the time needed to infer all possible pairs of program points that may happen in parallel.

Although the MHP analysis has been successfully applied to both industrial case studies, it has not been possible to run the industrial case studies due to limitations in the simulator which could not treat all parts of the application. Thus, there is no measure of error in these cases. We argue that the analyzer achieves high precision, with the approximated error less than 32.9% (bear in mind that $\mathbf{E_p}$ is a lower bound of the real parallelism) and up to 0% in other cases. As regards efficiency, both the method-level analysis and the graph construction are very efficient (just 0.36 sec. for the largest case study). The $\tilde{\mathcal{E}}_{\mathbf{P}}$ inference takes notably more time. But, as explained in Section 4.5 and illustrated in Section 5.3.3, for most applications only a subset of pairs is of interest and, besides, those pairs can be computed on demand.

Chapter 6

Conclusions, Related and Future Work

We have proposed a novel and efficient approach to infer MHP information for concurrent objects. The concurrent objects model is a innovative paradigm whose main focus is on exploiting the concurrency which is inherent to the concept of object. Our work constitutes, to the best of our knowledge, the first MHP analysis for this programming paradigm.

The main novelty of the analysis is that MHP information is obtained by means of a local analysis whose results can be modularly composed by using a MHP *analysis graph* in order to obtain global MHP relations. Besides, graph construction can be parametrized with a set of points of interest and MHP information can be computed on demand leading to a better efficiency for each particular use of the analysis.

6.1 Related Work

MHP analyses are highly dependent on the concurrency model and thus, it is difficult to compare technically the different existing analysis between each other.

Due to similarities between the concurrency models, the closest work to ours is the MHP analysis for X10 proposed in [13, 1]. We should first note that the async-finish model falls in the category of structured parallelism. This simplifies the inference of *escape* information, since the finish construct ensures that all methods called within its scope terminate before the execution continues to the next instruction. Moreover, it is important to note that our approach would achieve the same precision for their context-sensitive motivating example (Sec. 2.2 in [13]). This is because we do not merge calling contexts, but rather leave them explicit in the MHP graph. As additional advantages, by splitting the analysis in two phases we achieve:

- A higher degree of modularity and incrementally, since when a method is modified (or added, deleted, etc.), we only need to re-analyze that method locally, and replace its

corresponding sub-graph in the global MHP graph accordingly; and

- On demand MHP analysis, since we do not need to compute all MHP pairs in order to check if two given program points might run in parallel, but rather just check the relevant conditions for those two program points only.

An MHP analysis for Ada has been presented in [18], and extended and improved later for Java in [19]. Note that Ada has a *rendevouz* concurrency model (based on synchronous messages) while Java is based in threads and shared memory. In [18], they first construct a *trace flow graph* (TFG) which is a graph that combines all the *control flow graphs* (CFG) of each task and has special nodes that represent the synchronization points (where messages are exchanged) and that connect the different CFGs. Once the TFG has been created the analysis is defined through data flow equations. [19] provides a similar analysis for Java programs. The structure of the analysis is the same but *Parallel execution graphs* (PEGs) are generated instead of TFGs. As the concurrency model is different, they have different special nodes for synchronization. These nodes correspond to the concurrency primitives in Java (e.g. start(), join()) and their corresponding data flow equations are different as well.

The above work still has important limitations. Every thread has to be explicitly considered when constructing the PEG. Consequently, an upper bound in the number of threads is needed. This limitation has been overcome by [16, 9]. In [9], Java programs are abstracted to an *abstract thread model* which is then analyzed in two phases. This model can represent an unbounded number of threads which are distinguished by the creation point. That is, there is an abstract thread for each thread creation point (start()). MHP graphs are used as well despite being substantially different from ours.

A main difference between the previous set of approaches and ours is that we have a first phase to infer efficiently local information for each method, while they infer a thread-level MHP from which it is possible to tell which threads might *globally* run in parallel. In addition, unlike our method-level analysis, it does not consider any synchronization between the threads in the first phase, but rather in the second phase.

6.2 Future Work

An important application of MHP analysis is for understanding if two program points that belong to different tasks *in the same object* might run in parallel (i.e., interleave). We refer to this information as object-level MHP.

This information is valuable because, in any static analysis that aims at approximating the objects' states, when a suspended task resumes, the (abstract) state of the corresponding

object should be refined to consider modifications that might have been done by other tasks that interleave with it.

Our approach can be directly applied to infer object-level MHP pairs by incorporating points-to information [21, 17]. By incorporating this information we would obtain a context sensitive analysis in the sense of the one in [9]. Despite distinguishing threads by their creation point we could distinguish different objects by their allocation site.

The ABS language also supports **await** statements with boolean conditions (**await** c? suspends the execution until the condition c is satisfied). This statements can implement locks, barriers and other concurrency primitives. Thus, an interesting extension of the MHP analysis would be to take into account these statements. Such improvement could as well serve as a first step into adapting the analysis to other concurrency models.

Bibliography

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPOPP*, pages 183–193. ACM, 2007.

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

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *The 9th Asian Symposium on Programming Languages and Systems (APLAS'11)*, volume 7078, pages 238–254. Springer, 2011.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 151–154. ACM Press, January 2012.

[5] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. Analysis of may-happen-in-parallel in concurrent objects. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems*, volume 7273 of *Lecture Notes in Computer Science*, pages 35–51. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-30793-5_3.

[6] Elvira Albert, Antonio Flores-Montoya, and Samir Genaim. Maypar: A may-happen-in-parallel analyzer for concurrent objects. Technical report, 2012.

[7] J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[8] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *LCPC'05*, volume 4339 of *LNCS*, pages 152–169. Springer, 2005.

[9] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised*

Selected Papers, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2005.

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

[11] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[12] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Post Proc. of FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

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

[14] Jonathan K. Lee, Jens Palsberg, and Rupak Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.

[15] L. Li and C. Verbrugge. A practical mhp information analysis for concurrent java programs. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *LCPC'04*, LNCS, pages 194–208. Springer, 2004.

[16] Lin Li and Clark Verbrugge. A practical mhp information analysis for concurrent java programs. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*, volume 3602 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2004.

[17] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. In *ISSTA*, pages 1–11, 2002.

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

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

[20] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

Appendix A

Proofs

A.1 Proof of Corollary 4.4.2

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

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

- $\exists z \in P_{\mathcal{P}} : z \stackrel{i}{\rightarrow} x_1 \rightsquigarrow pi_1 \in \mathcal{G}_{P} \wedge z \stackrel{j}{\rightarrow} x_2 \rightsquigarrow pi_2 \in \mathcal{G}_{P} \wedge (x_1 \neq x_2 \vee (x_1 = x_2 \wedge i = j > 1))$. We only have to consider the case where $p = z$.

  In both cases, $x_1 \neq x_2$ and $(x_1 = x_2 \wedge i = j > 1)$, the set $\mathcal{L}_{P}(p)$ must have two atoms $y_1{:}m_1$ and $y_2{:}m_2$. If the condition holds there is $p' \in m$ such that $\mathcal{L}_{P}(p) \preceq \mathcal{L}_{P}(p')$ which implies that both atoms are covered by other atoms $y'_1{:}m'_1$ and $y'_2{:}m'_2$ in $\mathcal{L}_{P}(p')$. Therefore, we have that $p' \stackrel{i'}{\rightarrow} x'_1 \rightsquigarrow pi_1 \in \mathcal{G}_{P}$, $p' \stackrel{j'}{\rightarrow} x'_2 \rightsquigarrow pi_2 \in \mathcal{G}_{P}$ and $x'_1 \neq x'_2 \vee (x'_1 = x'_2 \wedge i' = j' > 1)\}$. That is, if p is a common ancestor of pi_1 and pi_2, then p' is one as well and thus p can be removed from $\mathcal{G}_{P}$.

A.1.1 Proof of theorem 4.5.1

In order to prove the soundness of MHP we extend the representation of program states and the corresponding semantics. The modified semantics is shown in Fig. A.1. Each task

contains additional information $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle$. $\mathcal{L}r$ is a set that records the calls that have been performed by the current task, and their status. It can be seen as a concrete version of $\mathcal{L}_P$. For each call, it contains information about the related future variable if there is one and whether the call might be running $\tilde{tid}$, yet to be started $\check{tid}$ or finished $\hat{tid}$.

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

Definition A.1.1. *Given a program P, we let $\mathcal{E}^r_P = \cup\{\mathcal{E}^r_S \mid S_0 \leadsto^* S\}$ where for $S = \langle O, Tk\rangle$, and $\mathcal{E}^r_S$ is defined as*

$$\mathcal{E}^r_S = \left\{ ((tid_1, pp(s_1)), (tid_2, pp(s_2))) \;\middle|\; \begin{array}{l} \langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle \in Tk_S, \\ \langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk_S, \\ tid_1 \neq tid_2 \end{array} \right\}.$$

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

We will show (1) that the $\mathcal{L}r$ sets contain the necessary concurrency information; (2) that $\mathcal{L}_P$ is an approximation of all possible $\mathcal{L}r$ sets; and (3) consequently Theorem 4.5.1 holds.

Definition A.1.2. *Given a state $S = \langle O_S, Tk_S\rangle$, we define a concrete graph $\mathcal{G}r_S$ using $\mathcal{L}r$ as follows*

$$\begin{array}{rl} \mathcal{G}r_S = & \langle V_S, E_S\rangle \\ V_S = & \{\tilde{tid}, \check{tid}, \hat{tid} | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S\} \\ & \cup cP_S \\ & \cup\{(tid, pp(s))_y | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S, y{:}x \in \mathcal{L}r\} \\ cP_S = & \{(tid, pp(s)) | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S\} \\ E_S = & ei_S \cup el_S \cup ef_S \\ ei_S = & \{\tilde{tid} \rightarrow (tid, pp(s)) | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S\} \\ & \cup\{\hat{tid} \rightarrow (tid, p_{\dot{m}}) | \langle tid, m, o, lk, l, \epsilon(v), \mathcal{L}r\rangle \in Tk_S\} \\ & \cup\{\check{tid} \rightarrow (tid, p_{\mathring{m}}) | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S, s = p_{\mathring{m}}, lk = \bot\} \\ el_S = & \{(tid, pp(s)) \rightarrow x | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S \wedge \star{:}x \in \mathcal{L}r\} \\ ef_S = & \{(tid, pp(s)) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow x \\ & | \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk_S \wedge y{:}x \in \mathcal{L}r\} \end{array} \tag{A.1}$$

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

Definition A.1.3.

$$\begin{array}{rl} \mathcal{EG}^r_S = & dMHP_S \cup iMHP_S \\ dMHP_S = & \{(x, y) | x, y \in cP_S \wedge x \leadsto y\} \\ iMHP_S = & \{(x, y) | x, y \in cP_S \wedge (\exists z \in cP_S : z \leadsto x \wedge z \leadsto y)\} \end{array} \tag{A.2}$$

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

$$(2)\quad \frac{\begin{array}{c}l(x) = oid_1 \neq \textsf{null}, l' = l[y \rightarrow tid_1], l_1 = buildLocals(\bar{x}, m)), tid_1 \text{is a fresh id}\\ ((oid_1 \neq oid \wedge \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}x/\star{:}x] + y{:}\widetilde{tid}_1) \vee (oid_1 = oid \wedge \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}x/\star{:}x] + y{:}\check{tid}_1))\end{array}}{\begin{array}{c}\langle O, \{\langle tid, m, oid, \top, l, y{=}x.m_1(\bar{x}); s, \mathcal{L}\mathbf{r}\rangle \parallel T\}\rangle \rightsquigarrow\\ \langle O, \{\langle tid, m, oid, \top, l', s, \mathcal{L}\mathbf{r}'\rangle, \langle tid_1, m_1, oid_1, \bot, l_1, body(m_1), \emptyset\rangle \parallel T\}\rangle\end{array}}$$

$$(3)\quad \frac{\langle oid, \bot, f\rangle \in O, O' = O[\langle oid, \bot, f\rangle / \langle oid, \top, f\rangle], v = l(x), \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}\check{x}/y{:}\tilde{x}]\}}{\langle O, \{\langle tid, m, oid, lk, \top, \textsf{return}\ x\rangle, \mathcal{L}\mathbf{r}\} \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \bot, l, \epsilon(v), \mathcal{L}\mathbf{r}'\rangle \parallel T\}\rangle}$$

$$(4)\quad \frac{l_1(y) = tid_2, \mathcal{L}\mathbf{r}'_{\mathbf{1}} = \mathcal{L}\mathbf{r}_{\mathbf{1}}[y{:}\widetilde{tid}_2/y{:}\widehat{tid}_2]}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, \textsf{await}\ y?; s_1, \mathcal{L}\mathbf{r}_{\mathbf{1}}\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v), \mathcal{L}\mathbf{r}_{\mathbf{2}}\rangle \parallel T\}\rangle \rightsquigarrow\\ \langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, s_1, \mathcal{L}\mathbf{r}'_{\mathbf{1}}\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v), \mathcal{L}\mathbf{r}_{\mathbf{2}}\rangle \parallel T\}\rangle\end{array}}$$

$$(5)\quad \frac{}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, lk, l_1, \textsf{await}\ y?; s_1, \mathcal{L}\mathbf{r}_{\mathbf{1}}\rangle \parallel T\}\rangle \rightsquigarrow\\ \langle O, \{\langle tid_1, m_1, oid_1, lk, l_1, \textsf{release}; \textsf{await}\ y?; s_1, \mathcal{L}\mathbf{r}_{\mathbf{1}}\rangle \parallel T\}\rangle\end{array}}$$

$$(6)\quad \frac{\langle oid, \bot, f\rangle \in O, O' = O[\langle oid, \bot, f\rangle / \langle oid, \top, f\rangle], \mathcal{L}\mathbf{r}' = \mathcal{L}\mathbf{r}[y{:}\check{x}/y{:}\tilde{x}]\}}{\langle O, \{\langle tid, m, oid, \top, \textsf{release}; s, \mathcal{L}\mathbf{r}\rangle \parallel T\}\rangle \rightsquigarrow \langle O', \{\langle tid, m, oid, \bot, l, s, \mathcal{L}\mathbf{r}'\rangle \parallel T\}\rangle}$$

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

$$(8)\quad \frac{l_1(y) = tid_2, l'_1 = l_1[x \rightarrow v], \mathcal{L}\mathbf{r}'_{\mathbf{1}} = \mathcal{L}\mathbf{r}_{\mathbf{1}}[y{:}\widetilde{tid}_2/y{:}\widehat{tid}_2]}{\begin{array}{c}\langle O, \{\langle tid_1, m_1, oid_1, \top, l_1, x{=}y.\textsf{get}; s_1, \mathcal{L}\mathbf{r}_{\mathbf{1}}\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v), \mathcal{L}\mathbf{r}_{\mathbf{2}}\rangle \parallel T\}\rangle \rightsquigarrow\\ \langle O, \{\langle tid_1, m_1, oid_1, \top, l'_1, s_1, \mathcal{L}\mathbf{r}'_{\mathbf{1}}\rangle, \langle tid_2, m_2, oid_2, \bot, l_2, \epsilon(v), \mathcal{L}\mathbf{r}_{\mathbf{2}}\rangle \parallel T\}\rangle\end{array}}$$

Figure A.1: *Extended semantics*

The following theorme express that $\mathcal{EG}^r_S$ captures the concurrency information of a given state S:

Theorem A.1.4. $\forall S : (S_0 \rightsquigarrow^* S) \Rightarrow (\mathcal{E}^r_S \subseteq \mathcal{EG}^r_S)$

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

Definition A.1.5. *We let the function* $\varphi : \mathcal{T} \times P_{\mathcal{P}} \rightarrow P_{\mathcal{P}}$ *be* $\varphi(id_1, p_1) = p_1$.

Definition A.1.6. $\mathcal{EG}_S = \{(\varphi(id_1, p_1), \varphi(id_2, p_2)) \mid (id_1, p_1, id_2, p_2) \in \mathcal{EG}^r_S\} = \{(p_1, p_2) \mid (id_1, p_1, id_2, p_2) \in \mathcal{EG}^r_S\}$

Theorem A.1.7. $\forall S : S_0 \rightsquigarrow^* S : \mathcal{EG}_S \subseteq \tilde{\mathcal{E}}_P$

Theorems A.1.4 and A.1.7 are proved in the next two sections. Once proved they imply the desired property:

$$\mathcal{E}_P = \varphi(\mathcal{E}_P^r) = \varphi(\cup_S \mathcal{E}_S^r) \overset{theorem A.1.4}{\subseteq} \varphi(\cup_S \mathcal{EG}_S^r) = \cup_S \varphi(\mathcal{EG}_S^r) = \cup_S \mathcal{EG}_S \overset{theorem A.1.7}{\subseteq} \tilde{\mathcal{E}}_P \quad \text{(A.3)}$$

A.2 Proof of Theorem A.1.4

Theorem A.1.4 is equivalent to:

$\forall S = \langle O, Tk\rangle : S_0 \rightsquigarrow^* S :$
$\forall \langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle, \langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk :$
$id_1 \neq id_2 : ((tid_1, pp(s_1)), (tid_2, pp(s_2))) \in \mathcal{EG}_S^r$

However, it is sufficient to prove that every task is reachable from the main node $((0, pp(s_0)))$ that corresponds to the main task $(\langle 0, \mathsf{main}, 0, lk_0, l_0, s_0, \mathcal{L}r\rangle)$. This can be expressed:

$\forall S : S = \langle O, Tk\rangle, S_i \rightsquigarrow^* S :$
$\exists \langle 0, \mathsf{main}, 0, lk_0, l_0, s_0, \mathcal{L}r\rangle \in Tk :$
$\forall \langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle \in Tk : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_1, pp(s_1))$

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

Base case: Straightforward. Only the main task is present. $\forall \langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle \in Tk : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_1, pp(s_1))$ trivially holds.

Inductive case: For any possible transition $S = \langle O, Tk\rangle \rightsquigarrow S' = \langle O', Tk'\rangle$. The induction hypothesis is:

$\exists \langle 0, \mathsf{main}, 0, lk_0, l_0, s_0, \mathcal{L}r\rangle \in Tk :$
$\forall \langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle \in Tk : tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_1, pp(s_1))$

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

1. Sequential step: The new state can be obtained through a substitution of the form $Tk' = Tk[\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle / \langle tid, m, o, lk, l', s', \mathcal{L}r\rangle]$ with the condition $pp(s) = p_{\mathring{m}} \rightarrow lk = \top$ and $s \neq s'$.

$\mathcal{G}r_{_{S'}} = \langle V_{S'}, E_{S'}\rangle$ and $\mathcal{G}r_{_S} = \langle V_S, E_S\rangle$ are isomorphic graphs and we can define a graph bijection as a substitution:

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

Where $y \in P_{\mathcal{F}}$. It is easy to see that the given substitution is indeed a bijection. Let $a \to b$ and edge of $\mathcal{G}r_{_S}$ we have one of the following:

(a) Both a and b are not $(tid, pp(s))$ nor $(tid, pp(s))_y$. In this case, $a \to b$ is in $\mathcal{G}r_{_{S'}}$ as they are not affected by the substitution.

(b) $a \to b = (tid, pp(s)) \to (tid, pp(s))_y$. This implies that $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in TK$ and $y{:}x \in \mathcal{L}r$. Then, $\langle tid, m, o, lk, l, s', \mathcal{L}r\rangle \in TK'$ with the same $\mathcal{L}r$ and $(tid, pp(s')) \to (tid, pp(s'))_y$ is in $\mathcal{G}r_{_{S'}}$.

(c) $a = (tid, pp(s))$, $b \neq (tid, pp(s))_y$. This implies that $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in TK$ and $\star{:}b \in \mathcal{L}r$. We have that $\langle tid, m, o, lk, l, s', \mathcal{L}r\rangle \in TK'$ with the same $\mathcal{L}r$ so $(tid, pp(s')) \to b$ is in $\mathcal{G}r_{_{S'}}$.

(d) $a = (tid, pp(s))_y$. This implies that $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in TK$ and $y{:}b \in \mathcal{L}r$. We have that $\langle tid, m, o, lk, l, s', \mathcal{L}r\rangle \in TK'$ with the same $\mathcal{L}r$ so $(tid, pp(s'))_y \to b$ is in $\mathcal{G}r_{_{S'}}$.

(e) $a \to b = \tilde{tid} \to (tid, pp(s))$. This implies that $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in TK$. We have that $\langle tid, m, o, lk, l, s', \mathcal{L}r\rangle \in TK'$. $\tilde{tid} \to (tid, pp(s'))$ is in $\mathcal{G}r_{_{S'}}$ by definition.

(f) $a \to b = \hat{tid} \to (tid, pp(s))$. This case is not possible, if $a = \hat{tid}$, $s = \epsilon(v)$ and in that case no rule can change s.

(g) $a \to b = \check{tid} \to (tid, pp(s))$. This case is not possible, this is because $a = \check{tid}$, $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in TK$ and $s = p_{\mathring{m}} \wedge lk = \bot$ which does not comply with the previously stated condition.

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

$$\exists\langle 0, \mathsf{main}, 0, lk_0, l_0, s_0, \mathcal{L}r\rangle \in Tk' : \forall\langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle \in Tk' :$$
$$tid_1 \neq 0, (0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_1, pp(s_1))$$

2. Release:
$Tk' = Tk[\langle tid, m, o, \top, l, s, \mathcal{L}r\rangle/\langle tid, m, o, \bot, l, s, \mathcal{L}r'\rangle]$ where $\mathcal{L}r' = \mathcal{L}r[y{:}\check{x}/y{:}\tilde{x}]$.

As the $\mathcal{L}r$ sets are always finite, without loss of generality we assume that only one element is substituted. If more than one elements were substituted the same reasoning could be applied repeatedly.

This change has no effect on the graph nodes, $V'_S = V_S$. However, it has an effect on the edges of the graph. By the graph definition we see that changes in a $\mathcal{L}r$ set affect the edges in el_S and ef_S.

- if $\star{:}\check{tid}_1$ is substituted by $\star{:}\tilde{tid}_1$:
  $el_{S'} = el_S \setminus \{(tid, pp(s)) \to \check{tid}_1\} \cup \{(tid, pp(s)) \to \tilde{tid}_1\}$ and $ef_{S'} = ef_S$.

  Given a task $\langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk$, by induction hypothesis, there exists $\langle 0, \mathsf{main}, 0, lk_0, l_0, s_0, \mathcal{L}r\rangle \in Tk$ such that $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_2, stmts_2)$. That is, there is a path p from $(0, pp(s_0))$ to $(tid_2, pp(s_2))$. A path can be seen as a finite string of edges of the graph $x_1 \to x_2, x_2 \to x_3 \cdots x_{n-1} \to x_n$.

  If $(tid, pp(s)) \to \check{tid}_1$ does not belong to p, then p is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$. Consequently, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.

  If $(tid, pp(s)) \to \check{tid}_1$ belongs to p. $p = x_1 \to x_2, x_2 \to x_3 \cdots (tid, pp(s)) \to \check{tid}_1, \check{tid}_1 \to (tid_1, pp(s_1)) \cdots x_{n-1} \to x_n$. We can create a new path $p' = x_1 \to x_2, x_2 \to x_3 \cdots (tid, pp(s)) \to \tilde{tid}_1, \tilde{tid}_1 \to (tid_1, pp(s_1)) \cdots x_{n-1} \to x_n$.

  This new path p' is valid in $\mathcal{G}r_{S'}$: $(tid, pp(s)) \to \tilde{tid}_1$ is the edge added in $el_{S'}$ and $\tilde{tid}_1 \to (tid_1, pp(s_1))$ belongs to both ei_S and $ei_{S'}$ by definition. In conclusion,$(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.
- if $y{:}\check{tid}_1$ $y \neq \star$ is substituted by $y{:}\tilde{tid}_1$: $ef_{S'} = ef_S \setminus \{(tid, pp(s))_y \to \check{tid}_1\} \cup \{(tid, pp(s))_y \to \tilde{tid}_1\}$ and $el_{S'} = el_S$.

  The same reasoning can be applied in this case as the origin of the substituted edge $(tid, pp(s))_y$ (or $(tid, pp(s))$ in the previous case) does not affect the proof.

3. Loss of a future variable association:
   $Tk' = Tk[\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle / \langle tid, m, o, lk, l, s, \mathcal{L}r'\rangle]$ where $\mathcal{L}r' = \mathcal{L}r[y{:}x/\star{:}x]$.

   It is easy to see that for a given future variable y there is at most one pair in $\mathcal{L}r$. If there is none, $Tk' = Tk$ and the property holds. Otherwise, one pair $y{:}x$ gets substituted by $\star{:}x$.

   This change makes a graph node disappear, $V'_S = V_S \setminus \{(tid, pp(s))_y\}$. As for the edges, by definition we see that changes in a $\mathcal{L}r$ set affect the edges in el_S and ef_S.

   $ef_{S'} = ef_S \setminus \{(tid, pp(s)) \to (tid, pp(s))_y, (tid, pp(s))_y \to x\}$
   $el_{S'} = el_S \cup \{(tid, pp(s)) \to x\}$

On the other hand, the loss of the lock could make new edges appear in ei_S but that cannot make any path disappear and thus affect the property.

Given a task $\langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk$, by the induction hypothesis we have that $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_2, s_2)$. That is, there is a path p from $(0, pp(s_0))$ to $(tid_2, pp(s_2))$.

The node $(tid, pp(s))_y$ has only one incoming and one outgoing edge. If any of those appears in p the other one has to appear in p as well.

If "$(tid, pp(s)) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow x$" does not belong to p, then p is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$. Consequently, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, stmts_2)$.

If "$(tid, pp(s)) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow x$" belongs to p, then $p = x_1 \rightarrow x_2, x_2 \rightarrow x_3 \cdots (tid, pp(s)) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow x \cdots x_{n-1} \rightarrow x_n$. We can create a new path $p' = x_1 \rightarrow x_2, x_2 \rightarrow x_3 \cdots (tid, pp(s)) \rightarrow x \cdots x_{n-1} \rightarrow x_n$.

This new path p' is valid in $\mathcal{G}r_{S'}$ as $(tid, pp(s)) \rightarrow x$ is the edge added in $el_{S'}$.Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.

4. New task added:

$$\begin{aligned} Tk' = \; & Tk[\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle / \langle tid, m, oid, lk, l, s, \mathcal{L}r'\rangle] \cup \\ & \langle tid_1, m_1, oid_1, \perp, l_1, body(m_1), \emptyset\rangle \end{aligned}$$

where $\mathcal{L}r' = \mathcal{L}r \cup \{y{:}\tilde{tid}_1\}$ or $\mathcal{L}r' = \mathcal{L}r \cup \{y{:}\check{tid}_1\}$.

$\mathcal{G}r_{S'} = \langle V', E'\rangle$ where $V' = V \cup \{\tilde{tid}_1, \check{tid}_1, \hat{tid}_1, (tid_1, p_{\mathring{m}_1}), (tid, pp(s))_y\}$ and $E' = E \cup \{(tid, s) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow \tilde{tid}_1, \tilde{tid}_1 \rightarrow (tid_1, p_{\mathring{m}_1}), \check{tid}_1 \rightarrow (tid_1, p_{\mathring{m}_1})$ or $E' = E \cup \{(tid, s) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow \check{tid}_1, \tilde{tid}_1 \rightarrow (tid_1, p_{\mathring{m}_1}), \check{tid}_1 \rightarrow (tid_1, p_{\mathring{m}_1})$.

In any case, $\mathcal{G}r_{S'} \supseteq \mathcal{G}r_S$ so any path in $\mathcal{G}r_S$ is still valid in $\mathcal{G}r_{S'}$. Applying induction hypothesis we conclude that for any task $\langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk$, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, s_2)$.

The only task that is in Tk' and is not in Tk is $\langle tid_1, m_1, o_1, \perp, l_1, body(m_1), \emptyset\rangle$. But the program point in this task is reachable from $\langle tid, m, o, lk, l, s, \mathcal{L}r'\rangle$ as we can create a path p from $(tid, pp(s))$ to $(tid_1, p_{\mathring{m}_1})$: $p = (tid, pp(s)) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow \tilde{tid}_1, \tilde{tid}_1 \rightarrow (tid_1, p_{\mathring{m}_1})$ or $p = (tid, pp(s)) \rightarrow (tid, pp(s))_y, (tid, pp(s))_y \rightarrow \check{tid}_1, \check{tid}_1 \rightarrow (tid_1, p_{\mathring{m}_1})$ are valid paths depending on the E' that we have.

We have already proved that $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid, pp(s))$ and $(tid, pp(s)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_1, p_{\mathring{m}_1})$. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_1, p_{\mathring{m}_1})$.

5. Task ending:

   $Tk' = Tk[\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \parallel \langle tid_1, m_1, oid_1, lk_1, l_1, \epsilon(v), \mathcal{L}r_1\rangle /$
   $\langle tid, m, oid, lk, l, s, \mathcal{L}r' \parallel \langle tid_1, m_1, oid_1, lk_1, l_1, \epsilon(v), \mathcal{L}r_1\rangle]\rangle$

   where $\mathcal{L}r' = \mathcal{L}r[y{:}\tilde{tid}_1 / y{:}\hat{tid}_1\}$.

   As previously stated, for a given future variable y there is at most one pair in $\mathcal{L}r$. If there is none, $Tk' = Tk$ and the property holds. Otherwise, one pair $y{:}\tilde{tid}_1$ gets substituted by $y{:}\hat{tid}_1$.

   This change has no effect on the graph nodes, $V'_S = V_S$. However, it has an effect on the edges of the graph. By the graph definition we see that changes in a $\mathcal{L}r$ set affect the edges in ef_S: $ef_{S'} = ef_S \setminus \{(tid, pp(s))_y \rightarrow \tilde{tid}_1\} \cup \{(tid, pp(s))_y \rightarrow \hat{tid}_1\}$

   Given a task $\langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk$, by induction hypothesis $(0, pp(s_0)) \overset{\mathcal{G}r_S}{\rightsquigarrow} (tid_2, pp(s_2))$. That is, there is a path p from $(0, s)$ to $(tid_2, pp(s_2))$.

   If $p_y \rightarrow \tilde{tid}_1$ does not belong to p, then p is a valid path in $\mathcal{G}r_{S'}$ as every edge in the path belongs to $E_{S'}$. Consequently, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, pp(s_2))$.

   If $p_y \rightarrow \tilde{tid}_1$ belongs to p, then $p = x_1 \rightarrow x_2, x_2 \rightarrow x_3 \cdots (tid, pp(s))_y \rightarrow \tilde{tid}_1, \tilde{tid}_1 \rightarrow (tid_2, p_{\dot{m}_2}) \cdots x_{n-1} \rightarrow x_n$. We can create a new path p' that is equal to $x_1 \rightarrow x_2, x_2 \rightarrow x_3 \cdots (tid, pp(s))_y \rightarrow \hat{tid}_1, \hat{tid}_1 \rightarrow (tid_2, p_{\dot{m}_2}) \cdots x_{n-1} \rightarrow x_n$.

   This new path p' is valid in $\mathcal{G}r_{S'}$ as $(tid, pp(s))_y \rightarrow \hat{tid}_1$ is the edge added in $ef_{S'}$ and $\hat{tid}_1 \rightarrow (tid_2, p_{\dot{m}_2})$ belongs to $\mathcal{G}r_{S'}$ and $\mathcal{G}r_S$ by definition. Therefore, $(0, pp(s_0)) \overset{\mathcal{G}r_{S'}}{\rightsquigarrow} (tid_2, pp(s_2))$.

6. Take lock:

   $Tk' = Tk[\langle tid, m, oid, \bot, l, s, \mathcal{L}r\rangle / \langle tid, m, oid, \top, l, s, \mathcal{L}r\rangle]$

   This transformation can make $\check{tid} \rightarrow (tid, p_{\mathring{m}} \in ei_S$ disappear in case $s = body(m)$ but it will not affect any path between program points due to theorem A.2.1. In order to apply this step $\langle oid, \top, f\rangle \in O$ which implies that any other task of the same object $\langle tid_1, m_1, oid_1, lk_1, l_1, s_1, \mathcal{L}r_1\rangle$ has not the object lock $lk_1 = \bot$ and, consequently, $y{:}\check{tid} \notin \mathcal{L}r$. Node $\check{tid}$ has no incoming edges in $\mathcal{G}r_S$ so there cannot be a path that goes through it.

Theorem A.2.1. $\forall S = \langle O, Tk\rangle : S_0 \rightsquigarrow^* S : \forall \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk(lk = \bot \rightarrow \nexists y{:}\check{x} \in \mathcal{L}r)$

Now the semantic rules are expressed as a combination of steps.

- Rule 1 and 5 are instances of step 1(Sequential step).
- Rule 2 is an instance of step 1 (Sequential step) followed by step 3 (Loss of future variable association) and step 4 (New task added).
- Rule 3 and 6 are step 1 (Sequential step) followed by step 2 (Release).
- Rule 4 and 8 are step 1 (Sequential step) followed by step 5 (Task ending).
- Rule 7 is an instance of step 6 (Take lock).

A.3 Proof of Theorem A.1.7

In order to prove Theorem A.1.7, we define the following function ψ.

Definition A.3.1. *ψ abstracts $\mathcal{L}r$sets into multisets in $\mathcal{B}$; ψ' abstracts a single mhp atom; and ψ'' abstracts tasks into methods.*

$$\psi''(\tilde{tid}) = \tilde{m} \tag{A.4}$$

$$\psi''(\check{tid}) = \check{m} \tag{A.5}$$

$$\psi''(\hat{tid}) = \hat{m} \text{ where } m = method(tid) \tag{A.6}$$

$$\psi'(y{:}x) = y{:}\psi''(x) \tag{A.7}$$

$$\psi(\mathcal{L}r) = \{(\psi'(a), i) | a \in \mathcal{L}r \wedge (\#i : b \in \mathcal{L}r : \psi'(a) = \psi'(b))\} \tag{A.8}$$

$$\tag{A.9}$$

Besides, an auxiliary lemma is needed:

Lemma A.3.2. $\forall S = \langle O, Tk \rangle : S_i \rightsquigarrow^* S : \langle tid, m, oid, lk, l, s, \mathcal{L}r \rangle \in Tk \Rightarrow \psi(\mathcal{L}r) \sqsubseteq \mathcal{L}_P(\varphi(tid, pp(s)))$

That is, the computed $\mathcal{L}_P$ is a safe approximation of the concrete property defined in the semantics. Knowing that we proceed to prove theorem A.1.7.

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

- $(x, x_1) \in dtMHP_S \Leftrightarrow x, x_1 \in cP_{\mathcal{S}} \wedge x \overset{\mathcal{G}r_S}{\rightsquigarrow} x_1$. That means there is a non-empty path $p = xa_1 \cdots a_n x_1$ expressed as a sequence of nodes in $\mathcal{G}r_S$.

By the graph definition we can infer a pattern (expressed as a regular expression) to which all non-empty paths from x to x_1 have to respond. "$l((l_y t + t)l)^{+}$" Where l, l_y, t are graph nodes, $l = (tid, pp(s))$, $t \in \{\tilde{tid}, \check{tid}, \hat{tid}\}$, $y \in P_{\mathcal{F}}$.

We prove that if $l(l_y t + t)l_1$ is a path in $\mathcal{G}r_S$, then there is a path $\varphi(l) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_1)$. Any other path is obtained by transitivity of $\overset{\mathcal{G}_P}{\rightsquigarrow}$.

Let $p = ltl_1$ where $t \in \{\tilde{tid}', \check{tid}', \hat{tid}'\}$, $l = (tid, pp(s))$, and $l_1 = (tid', pp(s'))$.

If p belongs to $\mathcal{G}_S$ then $\exists\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk : \star{:}t \in \mathcal{L}r$. We know that $a = \star{:}\psi''(t, i) \in \psi(\mathcal{L}r)$ with $i > 0$. By Lemma A.3.2 there is an atom $b \in \mathcal{L}_P(\varphi(tid, pp(s)))$, $b = (\star{:}h, j)$ with $j > 0$ and $h \in \{\tilde{m}', \check{m}', \hat{m}'\}$, such that $\star{:}h \succeq \star{:}\psi''(t)$.

Let $\mathcal{G}_P = \langle V, E\rangle$, We have that $\varphi(l), \varphi(l_1) \in P_{\mathcal{P}}$, which implies $\varphi(l), \varphi(l_1) \in V$. $(\varphi(l), h) \in E_2$ thanks to b. $(h, \varphi(l_1)) \in E_1$ as $\star{:}h \succeq \star{:}\psi''(t)$. In conclusion, we have that $\varphi(l) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_1)$ as we wanted to prove.

Lets consider the other option $p = ll_y tl_1$ where $t \in \{\tilde{tid}', \check{tid}', \hat{tid}'\}$, $l = (tid, pp(s))$, and $l_1 = (tid', pp(s'))$. If p belongs to $\mathcal{G}r_S$ then $\exists\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk : y{:}t \in \mathcal{L}r$. We know that $a = (y{:}\psi''(t), 1) \in \psi(\mathcal{L}r)$. By theorem A.3.2 there is an atom $b \in \mathcal{L}_P(\varphi(tid, pp(s)))$, $b = (y'{:}h, j)$ with $j > 0$. such that $y'{:}h \succeq y{:}\psi''(t)$.

Let $\mathcal{G}_P = \langle V, E\rangle$, We have that $\varphi(l), \varphi(l_1) \in P_{\mathcal{P}}$, which implies $\varphi(l), \varphi(l_1) \in V$. Thanks to b, if $y' \neq \star$, $(\varphi(l), \varphi(l)'_y), (\varphi(l)'_y, h) \in E_3$, otherwise $(\varphi(l), h) \in E_2$. $(h, \varphi(l_1)) \in E_1$ as $y'{:}h \succeq y{:}\psi''(t)$. In conclusion, we have that $\varphi(l) \overset{\mathcal{G}_P}{\rightsquigarrow} \varphi(l_1)$ as we wanted to prove.

By the definition of $\tilde{\mathcal{E}}_P$, $x' \overset{\mathcal{G}_P}{\rightsquigarrow} x'_1 \Rightarrow (x', x'_1) \in \tilde{\mathcal{E}}_P$.

- $(x, x_1) \in iMHP_S \Leftrightarrow x, y \in cP_{\mathcal{S}} \wedge \exists z \in cP_{\mathcal{S}}(z \rightsquigarrow x \wedge z \rightsquigarrow x_1)$. That is, we have two paths $p_1 = n_1 n_2 \cdots n_s x$ and $p_2 = n'_1 n'_2 \cdots n'_m x$ (where $n'_1 = n_1 = z$) expressed as a sequence of nodes in $\mathcal{G}r_S$.

  We take the shortest non-common suffix of p_1 and p_2. $p'_1 = n_j n_{j+1} \cdots n_s x$ and $p'_2 = n'_j n'_{j+1} \cdots n'_{m'} x_1$ such that $\forall i(0 < i \leq j : n_i = n'_i) \wedge n_{j+1} \neq n'_{j+1}$. Lets call $z' = n_j = n'_j$. We have that $z' \in cP_{\mathcal{S}}$ as in $\mathcal{G}r_S$ only program point nodes can have more than one outgoing edge. By the previously obtained result we have that $z'' = \varphi(z')$, $z'' \rightsquigarrow x'$ and $z'' \rightsquigarrow x'_1$ respect to $\mathcal{G}_P$.

  We also know that $n_{j+1} \neq n'_{j+1}$, which then implies that if $z' = (tid_2, pp(s_2))$, then $\exists\langle tid_2, m_2, oid_2, lk_2, l_2, s_2, \mathcal{L}r_2\rangle \in Tk : a = y{:}m, b = y'{:}m' \in \mathcal{L}r_2, a \neq b$. If $\psi'(a) = \psi'(b)$ we have $(\psi'(a), i) \in \psi(\mathcal{L}r_2)$, $i > 1$. Otherwise, $(\psi'(a), i), (\psi'(b), j) \in \psi(\mathcal{L}r_2)$, $i > 0, j > 0$.

By theorem A.3.2, either there is an atom $c \in \mathcal{L}_P(\varphi(tid, pp(s)))$, $c = (y'':m'', i')$ with $i' > 1$, such that $y'':m'' \succeq \psi'(a)$ and $y'':m'' \succeq \psi'(b)$ or two atoms $c_1, c_2 \in \mathcal{L}_P(\varphi(tid, pp(s)))$, $c_1 = (y'':m'', i'), c_2 = (y''':m''', j')$ with $i' > 0, j' > 0$, such that $y'':m'' \succeq \psi'(a)$ and $y''':m''' \succeq \psi'(b)$. Both cases imply that $(x', x'_1) \in \tilde{\mathcal{E}}_P$ as we wanted to prove.

A.4 Proof of Theorem A.2.1

$\forall S = \langle O, Tk\rangle : S_0 \rightsquigarrow^* S : \forall \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in Tk(lk = \bot \rightarrow \nexists y{:}\check{x} \in \mathcal{L}r)$

Base case The theorem trivially holds for S_0.

Inductive case We assume the theorem holds in the left side of each rule and see if it holds in its right side.

- Rules 1,4,5 and 8 do not change the state of the locks, therefore, applying induction hypothesis the theorem holds trivially.
- Rule 2 creates a new task with $lk = \bot$ but its $\mathcal{L}r = \emptyset$. The theorem holds.
- Rule 3 and 6 release the lock of $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in TK$ but at the same time, any $\check{x}$ in its $\mathcal{L}r$ set is substituted by $\tilde{x}$.
- Rule 7 obtains the lock which does not affect the property.
- Finally, the only rule that can add mhp atoms of the form $y{:}\check{x}$ is Rule 2 and requires the task to have the object lock. Any transition without the lock will maintain the property.

A.5 Proof of Theorem A.3.2

$$\forall S : S_i \rightsquigarrow^* S : \langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle \in S \Rightarrow \psi(\mathcal{L}r) \sqsubseteq \mathcal{L}_P(\varphi(tid, pp(s)))$$

When a task $\langle tid, m, oid, lk, l, s, \mathcal{L}r\rangle$ is created, $\mathcal{L}r = \emptyset$, $\psi(\mathcal{L}r) = \emptyset$, and $\mathcal{L}_P = (\varphi(tid, pp(s)) = \mathcal{L}_P(p_{\mathring{m}}) = \emptyset$ by definition. The transition function τ is equivalent to the transformations of $\mathcal{L}r$ performed in the semantics:

- the semantic rule 2 corresponds to cases (1) and (2) in τ;
- the semantic rules 3 and 6 correspond to case (4) in τ;

- the semantic rule 4 and 8 correspond to case (3) in τ;
- the semantic rules 1,5 and 7 do not change $\mathcal{L}r$ directly.

When branching occurs in rule 1 (**if** and **while** statements), the upper bound operation is applied to obtain a joint state that represents all possible branches. Rule 5 is taken into account thanks to the assumption done when applying τ that every **await** y? is preceded by a **release**.