MayPar: A May-Happen-in-Parallel Analyzer for Concurrent Objects [*]

Elvira Albert
Complutense University of Madrid
elvira@sip.ucm.es

Antonio Flores-Montoya
Complutense University of Madrid
aeflores@fdi.ucm.es

Samir Genaim
Complutense University of Madrid
samir.genaim@fdi.ucm.es

ABSTRACT

We present the concepts, usage and prototypical implementation of MayPar, a *may-happen-in-parallel* (MHP) static analyzer for a distributed asynchronous language based on *concurrent objects*. Our tool allows analyzing an application and finding out the pairs of statements that can execute in parallel. The information can be displayed by means of a graphical representation of the MHP analysis graph or, in a textual way, as a set of pairs which identify the program points that may run in parallel. The information yield by MayPar can be relevant (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 cost).

Categories and Subject Descriptors

F3.2 [**Logics and Meaning of Programs**]: Program Analysis; F2.9 [**Analysis of Algorithms and Problem Complexity**]: General; D.1.3 [**Programming Techniques**]: [Concurrent Programming] *Distributed programming, Parallel programming*

General Terms

Languages, Theory, Verification, Reliability

Keywords

Static Analysis, Resource Guarantees, Parallelism, Concurrent Objects

[*]This demo illustrates the implementation of a technique presented at FMOODS/FORTE 2012 [3]. The 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 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

1. INTRODUCTION

It is widely recognized that writing concurrent programs is error-prone. The concurrency errors that programmers face are frequently related to undesired task interleavings, which may be the result of a wrong synchronization of tasks. For instance, it frequently happens that, when a task suspends its execution, another task starts to execute and possibly modifies the global state. Without the programmer expecting it, the global state has changed when the suspended task resumes. This type of synchronization errors can lead to unpredicted exceptional behaviours, to non-termination, to erroneous computed results, etc.

This paper reports on a MAY-happen-in-PARallel analyzer for a distributed asynchronous language based on *concurrent objects*. The goal of a may-happen-in-parallel analysis is to identify *pairs* of statements that can execute in parallel (see, e.g., [7]). If the analysis does not infer an MHP pair for two points p_1 and p_2, it is guaranteed that p_1 and p_2 will never execute in parallel. This information provides a global perspective of the communication and synchronization among objects and enables better comprehension of the tasks interleavings that might occur along the program execution.

2. DESCRIPTION OF MayPar

MayPar can be downloaded from its website [1] as a self-contained binary executable (currently only available for Linux systems). It can also be used from a web interface that allows users to try out the system without installing it.

MayPar analyzes programs written in ABS [6], an *actor*-based language which has been recently proposed to model distributed concurrent systems. For many application areas, standard mechanisms like threads and locks are too low-level are considered to be error-prone and not modular enough. The concurrent objects model is based on considering objects as the concurrency units, i.e., each object conceptually has a dedicated processor and can run in parallel with other objects. Communication is based on asynchronous method calls with standard objects as targets. An essential difference with thread-based concurrency is that switching between tasks of the same object happens only at specific scheduling points during the execution (namely at await points), which are explicit in the source code and can be syntactically identified. This feature is essential for the precision of our MHP analysis.

[1]`http://costa.ls.fi.upm.es/costabs/mhp`

2.1 Main Concepts

Automatically finding MHP relations is challenging. Consider for example the following piece of code:

```
1 int m()  {                    6 int p() {
2   x.p();                      7   x.r();
3   f=x.q(); await f?;          8   ...
4   ...                         9 }
5 }                            10
```

where x is an object, f is a future variable used to synchronize with the result of the asynchronous call x.q, such that the instruction await allows releasing the processor if the task x.q has not terminated. An MHP relation (p_1, p_2) is *direct* if, when a task t_1 is running at program point p_1, there is another task t_2 executing p_2, and t_1 has triggered (transitively) the execution of t_2, or vice versa. For instance, the instructions of method p (from program point 8 on) and those of r may run in parallel and have a direct relation. The second type of *indirect* MHP relations (p_1, p_2) happen when a task t_3 is running and there are two other tasks t_1 and t_2 executing in parallel, respectively, at p_1 and p_2, which have been both triggered (transitively) from t_3 but which do not have a transitive relation between them (they do not trigger one another). This is the case of all instructions of p and q, they may run in parallel since they are both called from m.

In order to obtain global MHP information (i.e., all points in the program that can potentially run in parallel), the analyzer starts from a method (typically main) and computes an *MHP analysis graph* by analyzing all code which is reachable from the selected method. The technical description of how the graph is computed can be found in the paper [3]. From the graph, the MHP pairs of interest can be computed *on demand* by querying the system with the set of points the user wants to obtain information for. If only one point is given, the analysis gives all program points that can run in parallel with it. When a set of points is provided, our tool computes the MHP relations among them. This contrasts with other approaches (e.g, [1]) which define an analysis which necessarily computes *all* MHP relations.

2.2 Applications

MHP relations are useful to understand the concurrent behaviour of the application. As we will see in the tool demonstration, unexpected MHP relations can spot bugs in the programs which are often related to a bad synchronization of the involved tasks. Therefore, in this sense, MayPar is a useful debugging tool which allows the user to better understand the concurrent behaviour of his program.

A main motivation behind the development of MayPar is to infer non-functional properties of the program, such as termination and resource usage. For instance, given a simple loop of this form: “while (l!=null) {f=async(l.data); await f?; l=l.next;}”, in order to prove termination of the loop, we need to prove that the execution of the loop cannot run in parallel with any instruction that increases the length of the list l. Also, in order to bound its resource consumption, we need to bound the number of iterations of the loop. If the length of the list is not increased, the upper bound length(l) can be automatically found.

3. RELATED TOOLS

Multiple MHP analyses for different languages have been proposed, namely [7, 1] for the X10 language and several MHP analyses for Java in [9, 8, 5]. Our tool is an implementation of the MHP analysis for ABS programs proposed in [3], where a detailed technical comparison with previous analyses is provided. We herein focus on comparing MayPar with existing implementations. The algorithm proposed by [9] resulted in an implementation which is integrated as part of the FLAVERS tool for data flow analysis of concurrent programs. As regards [5], though they seem to have a working implementation on which they run some benchmarks, as far as we know, it is not available nor integrated in another tool. Similarly, [10] evaluate their analysis for OpenMP C programs but do not have an available implementation. Also, [4] presents a static schedule analysis of fine-grained parallelism with explicit happens-before relationships. Its approach is radically different as the scheduling has to be explicitly defined by the programmer. Besides, despite having an available implementation, it does not provide any user interface. Therefore, to the best of our knowledge, MayPar is the first standalone MHP analyzer which can be used for debugging purposes and that provides a user-friendly interface with several options to interact with the user. Besides, it is integrated as part of the COSTABS analyzer [2], where plays an essential role to prove the termination of concurrent programs and infer its resource consumption.

4. REFERENCES

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *PPOPP'07*, pages 183–193. ACM Press.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *PEPM'12*, pages 151–154. ACM Press.

[3] E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of may-happen-in-parallel in concurrent objects. In *FMOODS/FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer.

[4] C. M. Angerer and T. R. Gross. now happens-before later: static schedule analysis of fine-grained parallelism with explicit happens-before relationships. In *SPLASH '10*, pages 3–10. ACM.

[5] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *LCPC'05*, volume 4339 of *LNCS*, pages 152–169. Springer.

[6] 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.

[7] J. K. Lee and J. Palsberg. Featherweight X10: A Core Calculus for Async-Finish Parallelism. In *PPoPP'10*, pages 25–36. ACM Press.

[8] L. Li and C. Verbrugge. A practical mhp information analysis for concurrent java programs. In *LCPC'04*, LNCS, pages 194–208. Springer.

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

[10] Y. Zhang, E. Duesterwald, and G. Gao. Concurrency analysis for shared memory programs with textually unaligned barriers. In *LCPC 2007*, volume 5234 of *LNCS*, pages 95–109. Springer.

```
1  class User(String email) {
2    List<String> msgs;
3    User receive(String m) {
4      msgs = Cons(m,msgs);
5    }
6  }
7
8  class AddrBook(List<User> users) {
9    User getUser(String email) {
10     ...
11   }
12 }
13
14 class Notifier(AddrBook ab) {
15   List<String> addrs = Nil;
16   Unit notify(String m) {
17     while ( addrs != Nil ) {
18       Fut<User> u;
19       u = ab!getUser(head(addrs));
20       await u?;
21       User us = u.get;
22       us!receive(m);
23       addrs = tail(addrs);
24     }
25   }
26
27   Unit addAddr(String u) {
28     addrs = Cons(u, addrs);
29   }
30
31   Unit addAddrs(List<String> l) {
32     while ( l != Nil ) {
33      addAddr(head(l));
34      l=tail(l);
35     }
36   }
37 }
38
39 main{
40   User u1 = new User("a@b.com");
41   ...
42   AddrBook ab = new AddrBook([u1,...]);
43   Notifier ms = new Notifier(ab);
44   Fut<Unit> x = ms!addAddrs(["a@b.com",...]);
45   await x?
46   ms.notify("Hello ...");
47 }
```

Figure 1: The Notifier example

APPENDIX

A. USAGE OF MayPar

The following example demonstrates the behavior and main use of the tool: aiding the programmer in debugging and understanding the concurrent behaviour of programs;

A.1 The Notifier Example

We use the program depicted in Fig. 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 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. 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 the registered users with a 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: (a) it creates several User objects at L40-41, each with a unique email address; (b) it creates an AddrBook object at L42, and passes it a list of users [u1 ,...] ; (c) it creates a Notifier object at L43 which receives the address book ab as class parameter; (d) it adds some email addresses to be notified by asynchronously calling addAddrs at L44, and waits at L45 until it has terminated; and (e) finally it calls method notify at L46 in order to notify all registered users with a given message.

A.2 MHP Analysis of the Notifier example

The *input* to the tool is an ABS program, a selection of the method from which the analysis starts, and optionally a set of program points of interest. These program points can be referenced through program line numbers.

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.

Let us demonstrate the output of MayPar on the program of Fig. 1. The corresponding MHP graph is depicted in Fig. 2. 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 ⓘ. These nodes always include the method's *entry* and *exit* program points. In principle, other program points can be included, however, these are the only ones required for soundness. Each method m contributes two nodes: $\boxed{m}$ represents an instance of m that is *active*, i.e., running and

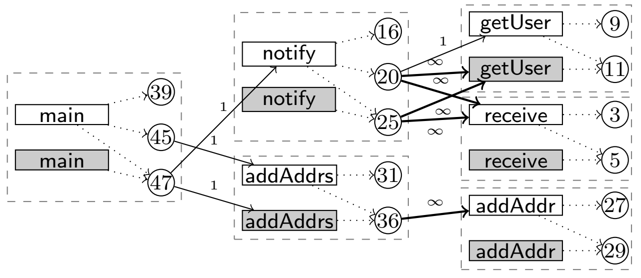

Figure 2: MHP graph for the program of Fig. 1

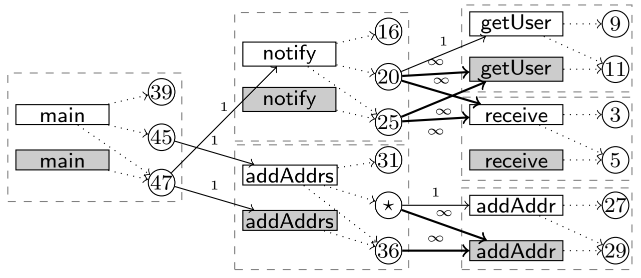

Figure 3: MHP graph after correcting addAddrs

can be at any program point, and $\boxed{\text{m}}$ represents an instance of m that is *finished*, i.e., it is at the exit program point.

The MHP graph is composed of 5 subgraphs, one for each method and that are represented as dashed rectangles. 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 $\boxed{\text{main}}$ to nodes ㊴, ㊺, and ㊼; and from $\boxed{\text{main}}$ to ㊼.

The subgraphs are interconnected 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). These edges are inferred by applying a *method-level* MHP analysis which analyzes each method separately. This analysis infers, for each program point, which methods might be running in parallel with that program point, how many instances of each, and in which mode (active or finished). This information is inferred by considering only the code of the corresponding method. For example, the method-level analysis infers: (a) for method main, at L45, there might be one active instance of method addAddrs. This will add an edge from ㊺ to $\boxed{\text{addAddrs}}$. The edge is labelled with 1 to indicate that it is only one instance of addAddrs; and (b) 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 $\boxed{\text{getUser}}$ with label 1, to $\boxed{\text{getUser}}$ with label ∞, and to $\boxed{\text{receive}}$ with label ∞. Edges with ∞ should be interpreted as infinitely many edges with weight 1.

The MHP graph guarantees that if there is an execution in which the instructions at program points i and j might execute in parallel, at least one of the following holds:

- *direct relation*: there is a path from ⓘ to ⓙ (or vice versa); or
- *indirect relation*: there is a node ⓚ that has two *different* paths to both ⓘ to ⓙ.

These properties allow generating all MHP pairs from the graph, or providing 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 induces 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). We will see that 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.

A.3 Using MHP for debugging and understanding concurrent programs

While testing the program of Fig. 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 allow 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. 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. 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. 3, where ⋆ corresponds to the new program point. Now we can see that the indirect MHP that involve ⑳ and ㉗ has been eliminated.

It is important to note that the construction of the MHP graph of Fig. 3 can be done incrementally from that of Fig. 2. We only reanalyze method addAddrs and generate a new subgraph for this method, the rest remain the same. This is because, as we have explained above, the construction of the MHP graph is done by analyzing each method separately. Thus, the change in addAddrs will not affect the subgraphs of other methods.