COSMOPEN:
Dynamic reverse-engineering on a budget

Technical Report COMP-002-2008
Computing Department
Lancaster University

Francois Taiani*, Marc-Olivier Killijian, Jean-Charles Fabre****
*** Computing Department, Lancaster University, Lancaster, UK**
**** LAAS-CNRS, Toulouse, France**

Abstract:

In this article we present COSMOPEN, a reverse engineering tool optimised for the behavioural analysis of complex layered software. COSMOPEN combines cheap and non-intrusive observation techniques with a versatile graph manipulation engine. By programming different graph manipulation scripts, the "focal length" of our tool can be adapted to the different abstraction levels found in complex software. We illustrate how our tool can be used to extract high-level behavioural models from a complex multithreaded platform (GNU/Linux, CORBA middleware).

Keywords: reverse engineering, middleware, model

1 INTRODUCTION

Mission-critical applications are increasingly assembled from third party components whose quality can often only partially be controlled (e.g. Commercials Off the Shelf (COTS), Free and Open-Source Software (FOSS)) [Mitre03, Stolper99]. This results in large and complex software platforms that are difficult to assess and analyse. To guarantee the overall dependability of their application, developers must therefore find ways to improve their system's robustness, usually by adding fault-tolerance mechanisms. This kind of extension, however, requires a thorough understanding of how a system's internal components behave and interact. To implement most fault-tolerance mechanisms, developers must for instance be able to track causal dependencies, to allow them to control non-determinism, analyse state entangling, and capture and restore consistent application states, just to name a few issues. To reach these levels of comprehension, programmers must often reverse-engineer the software they use in order to obtain trustworthy and up-to-date information that is detailed enough to carry out the needed analysis and adaptation.

Software reverse engineering has been intensively studied in the past, resulting in a wide range of approaches, both for structural and behavioural analysis. A major challenge these techniques have had to tackle arises from the sheer complexity of the data obtained when observing a program, in particular at run-time. Complex software systems often contain many layers (OS kernel, system libraries, middleware, GUI, *etc.*), each obeying its own internal logic. Each layer interacts with its neighbours in specific and hidden ways. When observing a complex system, these layers tend to overlap, resulting in a blurred image in which different levels of abstraction co-exist in the same data, a situation we refer to as *cross-layer entangling*. While tools exist that help navigate the complex data spaces that are obtained [Ebert02, Chen95, Frohlich94], the task generally remains cumbersome, time-consuming, and error-prone when applied to complex multi-layer platforms.

A second challenge faced by dynamic reverse engineering (i.e. using run-time data) is caused by the cost of dynamically obtaining fined-grained information about a program's execution. Exhaustive tracing, while optimal in terms of coverage, very rapidly becomes intractable on the large and long-running systems we consider here.

To address these two challenges, this paper proposes a dynamic reverse engineering approach that helps developers construct high-level behavioural models of complex multi-layered systems, while minimising observation costs. Our prototype, COSMOPEN[1], combines a cheap and non-intrusive observation technique based on partial observation, with a simple yet powerful scripting language for graph transformation. Because it minimises intrusion and relies on a flexible analyser, our technique is applicable to a wide range of industry-grade platforms, which we illustrate by reporting on the behavioural analysis of commercial multi-threaded CORBA ORBs that we've performed in the context of fault-tolerance hardening.

In addition to being an experimental report on the practice of reverse engineering, this paper highlights the possible trade-off between observation cost and model accuracy when reverse-engineering complex platforms. More precisely we show that even with an almost minimal observation strategy, useful and relevant information can be extracted from running programs by relatively simple means. We show that our approach is scalable both in terms of performance and complexity and can be applied on large-scale industry-grade complex platforms.

[1] Available on-line under GPL licence at http://ftaiani.ouvaton.org/7-software/

The remainder of our article is organised as follows: We first discuss in more detail the challenges raised by the dynamic reverse engineering of large multi-layer software in Section 2. We then detail two motivating examples that illustrate COSMOPEN's core capabilities and principles (Section 3). We move on to describe COSMOPEN's implementation and transformation language (Section 4). Section 5 present results we have obtained in the context of fault-tolerant computing on three popular Object Request Brokers (ORBs): ORBACUS, OMNIORB, and TAO, with a detailed case study of ORBACUS. We briefly review related work in Section 6, and conclude in Section 7.

2 Problem Statement

Our interest in reverse engineering tools originated from our work on fault-tolerance provisioning in complex software platforms [Taïani03, Taïani05, Killijian00, Pérennou98]. For economic reasons, pre-existing software components (OS, libraries, virtual machine, middleware) are increasingly used in application domains with high dependability requirements (railways, avionics, automobile, space exploration, communication), causing the developers of such systems to face the following two key challenges:

1. Because most available components are not specifically developed with fault-tolerance in mind, extra mechanisms are required to harden the resulting systems, and preclude any catastrophic failure. Dependability being a property of the overall system, these mechanisms demand a holistic approach to understand how each component's behaviour can threaten the system's reliability, and symmetrically contribute to the overall robustness of the emerging system.
2. Due to the complexity of modern system development, a high degree of separation between functional aspects and dependability related concerns is required. Developer teams must be allowed to address fault-tolerance mechanisms without interfering with the functional development of the remaining system.

We have addressed those challenges by adapting a well-known architectural paradigm called *computational reflection* to the specificities of large and complex systems. We found out that a key step to address the above points is to precisely understand how each component contributes to the overall system properties. This understanding can then be used to identify the observation and control points required to harden the system. Unfortunately, understanding how a component can influence the dependability of a larger system requires a precise analysis of this component internal behaviour and structure, an extremely complex task. In the context of our work, we wanted to understand how each component could interfere with the system's determinism, and how the state information related to one process was scattered throughout the platform's layers (OS kernel, system libraries, middleware, database management system, application). Our experimental targets were industry grade systems that relied heavily on multithreading and typically contained more than 100,000 lines of codes (C and C++). This led us to look for a reverse engineering approach that would be:

Dynamic:

While structural information is needed, our main interest lies in the dynamic behaviour of the system.

Non-intrusive:

We do not want to instrument components and libraries, to avoid costly customisation, and seamlessly support the use of COTS (Commercial Off The Shelf).

Cheap:

Because of the size and complexity of the systems we considered (more than 100,000 lines of code per components), we wanted to be able to observe them with a fine granularity, while maintaining an acceptable performance.

Flexible:

The approach should be able to adapt its granularity of observation during the different phases of a program's execution. This was needed because we were essentially interested in certain specific phases of a program run, and not in others. In the case of CORBA, we did not want to observe the middleware initialisation phase in the same details as the request handling mechanism.

Discriminative:

The approach had to be able to tackle cross-layer entangling and discriminate between different logical planes of a system's execution. For instance, we needed to be able to focus on the request management inside the middleware level, while abstracting away from the fine-grained thread activities related to communication management.

None of the reverse-engineering approaches proposed so far seemed to fit all these requirements. Traditional reverse-engineering tools can be roughly classified in two categories, none of which exactly matched our needs: (1) tools for structural analysis that use graph manipulation do not scale well to behavioural data, while (2) tools for behavioural analysis that employ aggregative techniques tend to be unsuitable for program comprehension (Figure 1).

The first category relies on graph operations and navigation to help a user encompass the structural complexity of a program. CIAO [Chen95,Chen97,Emden00], GUPRO [Ebert02], RIGI [Wong95,Wong98], are prime examples of this category.

By contrast, and because behavioural data are inherently more voluminous than structural ones (a single function can be invoked multiple times), tools in the second category use aggregative metrics (adding up the times a processor spends in a given function for instance), to present quantitative data about a program's execution in a condensed form. This kind of aggregation can be seen as a collapsing of the time axis onto the spatial structure of the program. Classical performance profilers, like `gprof` [Graham83], are a typical example of this approach. `gprof` uses an instrumented version of a program to gather information about the execution times for each of its functions. It then propagates those times along the edges of the call graph, and displays the resulting information under different textual formats. Quantitative aggregation of execution data is very useful to identify performance bottlenecks (hot spots) [Ottogalli01], or to profile resource consumption [Reiss00], but it does not convey much insight about a program's internal logic, as a plain call graph often removes too much information to be of any use in understanding a complex software system [Jerding97].

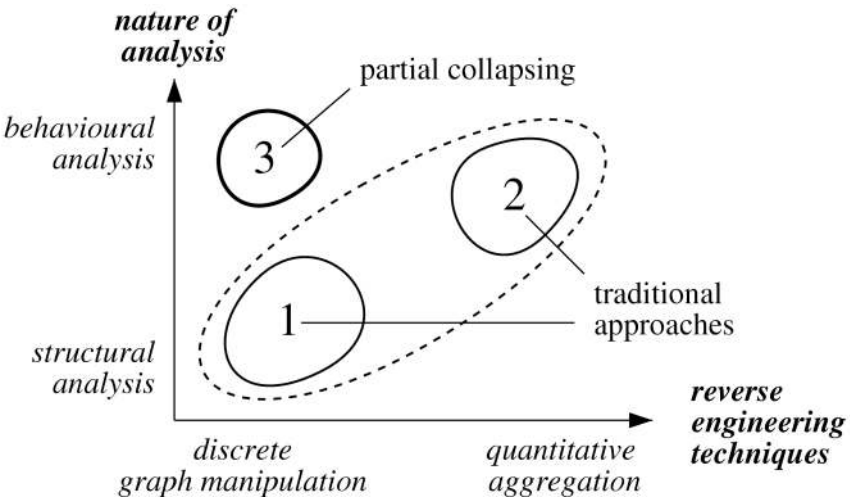


Figure 1: A classification of reverse engineering approaches

Graph manipulation has not traditionally been used for behavioural analysis because, without proper abstraction mechanisms, the dynamic events observed in a large system are too numerous to be represented intuitively as a discrete structure. Only recently have pattern-based approaches been proposed to represent large behavioural data set as graphs, and thus overcome this complexity lock. The base idea, which we will term *partial collapsing*, is to use behavioural patterns to "dose" the collapsing of the behavioural information into a more compact representation (Figure 1). By selectively folding together recurring patterns in the execution, these approaches decrease the complexity of the data to be represented, while retaining enough information to capture the program's internal logic. Jerding *et al.* for instance proposed a pattern extraction technique that collapses identical subtrees in the original call tree, and identifies duplicated subtrees generated by iteration and recursion [Jerding97]. This "pattern-induced" collapsing of subtrees has been also used by Pauw *et al.* to visualise Java reference graphs to help locate memory leaks in Java programs. Their technique groups program objects according to their class *and* the actual objects they refer to [Pauw00]. By compacting reference relationships into patterns, they help the user encompass a higher degree of complexity, but maintain enough information to discriminate groups of objects that do not correspond to the same "abstract" situation of referencing (i.e. they do not have the same pattern of references).

In this paper we propose a new partial collapsing techniques that allows graph manipulation to be applied to the behavioural analysis of complex multi-layer platforms. Unlike the pattern-based approaches we have just described, we do not use the recurring patterns induced in execution traces by local programming structures (loops, recursions, *etc.*), but takes advantage of the architectural structures found in complex software systems. The prototype in which we have implemented our approach (COSMOPEN) combines two key techniques: (1) it uses a non-intrusive, and inexpensive event extraction scheme, which specifically targets macroscopic interactions within a complex system; (2) it implements an imperative graph manipulation language, which allows users to easily focus on the level of abstraction they are interested in. In a way, this graph manipulation language acts as a *logical lens* that adapts the focal length of our tool to produce a crisp view of a system's dynamics from an blurred set of raw data.

The general philosophy of our tool can be compared with the one behind modern ground telescopes [O'Byrne96]. Because they stay on the ground, these telescopes are far cheaper than space launched systems like the Hubble Space Telescope. However, in order to overcome the blurring effect caused by Earth's atmosphere, they must deploy complex computer-based

techniques (speckle interferometry, adaptive optics, *etc.*) to reconstruct sharp pictures. In those telescopes, as in COSMOPEN, a "cheap" observation approach is balanced by using advanced software intelligence to produce sharp views of masked phenomena (be it by distance of by architectural complexity).

3 Introducing COSMOPEN: Two Motivating Examples

To motivate our problem and illustrate the key intuition behind our work, we present here two examples of behavioural reverse engineering. The first example is purely invented and has been kept very basic for ease of exposition. It illustrates the strategy we have developed to lower observation costs while maximising the yield of relevant behavioural data. The second example addresses more complex issues by considering a small multithreaded program. In this second example the collected behavioural data contains entangled elements from both the Linux multithreading library and the program itself. We show how COSMOPEN is able to untangle those distinct logical planes using graph transformation. Before we present these examples, we need however to give a brief presentation of COSMOPEN's main architecture and functions.

3.1 COSMOPEN's Overall Architecture

COSMOPEN's architecture follows the general guidelines proposed by Chen *et al.* for large source code repositories [Chen95] (Figure 2):

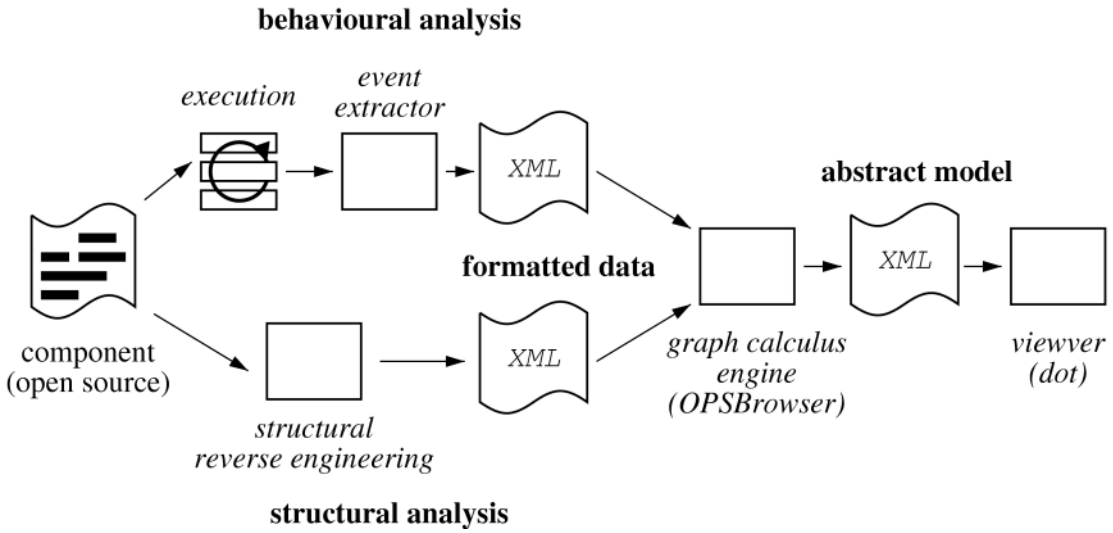


Figure 2: COSMOPEN's general architecture

1. Raw observation data is extracted from the target program. COSMOPEN can handle structural as well as behavioural data, using different extractors. This article focuses exclusively on the behavioural features of COSMOPEN .
2. The raw data is translated into a machine-friendly XML dialect.
3. COSMOPEN's graph calculus engine is used to construct high-level models of the observed data. Its specific operators and its imperative programming features allow the construction of arbitrary complex filters that are well adapted to the complex layered structures commonly found in industry-grade software.
4. The obtained information is presented to the user using an appropriate external viewer. We use the graph layout engine `dot` from AT&T [Graphviz07].

3.2 Controlling Observation Costs: A Basic Example

The problem

This first example introduces the basics of our tool COSMOPEN, and details our approach to limit observation costs. We will consider here an imaginary broadcasting middleware. The middleware is a third party open-source component and in order to implement additional fault-tolerance mechanisms, the project developers need to understand how broadcast requests are processed. They know nothing about the middleware's implementation, or of its overall structure, except that it provides a simple `broadcast` primitive and uses the `send` primitive provided by the underlying OS (Figure 3).

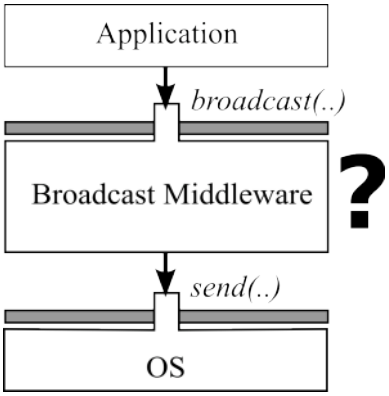


Figure 3: A simple multi-layer reverse-engineering problem

One way to obtain behavioural information about the broadcast middleware could be to trace all function and method invocations at run-time. Many tools such as open-compilers [Tatsubori00], binary code manipulators [Chiba03, Schwarz02], or aspect-oriented systems can be used to this aim. For portability and transparency reasons, we rely in COSMOPEN on the tracing capabilities provided by modern debuggers such as `gdb` [Stallman02]. Tracing all function/method invocations is however extremely costly in terms of performance. For the CORBA products we consider in our case studies (Section 5), the middleware would not even execute properly when traced at this level of granularity due to timeout watchdogs.

Foliage and rootage

Avoiding intractable observation overheads in large software demands a more balanced approach than exhaustive tracing. The strategy we have developed in COSMOPEN relies on the following two observations:

1) **Stack-Trace Captures:** Most tracing frameworks, and debuggers in particular, allows the capture of stack-traces each time a function/method is intercepted. A stack trace contains the pending calls that led to this particular function or method to be invoked, and therefore provides control flow information about *multiple* invocations at the cost of only one interception. This key insight is extremely valuable to lower tracing overheads in large systems, and forms the basis of COSMOPEN's event extractor component.

2) **Minimised observation footprint:** Although the developers of our example do not know anything about the middleware implementation, the interfaces by which the middleware component interacts with the rest of the system are known and well-documented (in our toy example `send` and `broadcast`). Combined with stack-trace captures, these interfaces can be used as entry-points to limit observation to a relevant subset of invocations related to the middleware activity in the system. We call this subset *an observation footprint*. The rationale behind this notion is represented on Figure 4 below. In a multilayer system, a

layer (an Object Request Broker, an OS kernel) acts as a broker between its surrounding upper and lower layers. It imports and uses lower-level programming primitives to implement and export higher-level programming entities. Using a biological metaphor, imported primitives form a *foliage pattern* inside the layer implementation, while exported interfaces are *rooted* into it (Figure 4). By capturing stack traces on a of well-chosen observation footprint we can follow this "roots-and-branches" structure and reconstruct a view of a layer's internal workings.

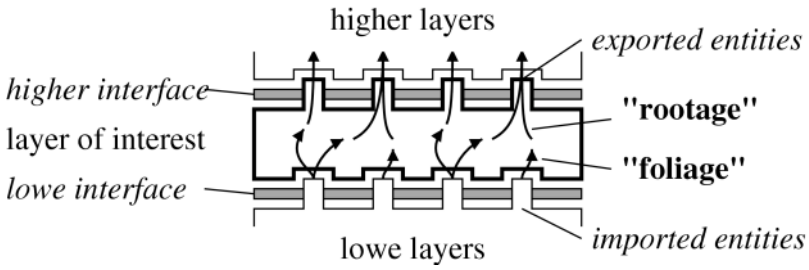


Figure 4: Leafage and Rootage in multi-layer system

Our approach combines the two above points by restricting our observation footprint to the methods and functions located on the upper and lower interfaces of a particular layer. In the above example, these are `broadcast`, `send`, plus any other OS system calls the middleware might be using, for instance for synchronisation or memory management (we return to this last point below). Let's assume that developers first wish to know how the middleware interacts with the OS's network API: We can here restrict our observation to `send`. Whenever an invocation to `send` is intercepted, COSMOPEN will capture the stack of the active thread (i.e. the set of pending calls for this thread), and thus obtains control flow information about additional methods and functions in the system. In our toy example, the first interception with `gdb` of `send` yields the following trace (simplified `gdb` output):

```
#0 send () at OS.cpp:18
#1 0x10000b94 in BroadcastEngine::marshallAndSend() at middleware.cpp:21
#2 0x10000bf4 in BroadcastEngine::broadcast() at middleware.cpp:26
#3 0x1000080c in Application::launch() at main-application.cpp:23
#4 0x1000078c in main() at main-application.cpp:30
```

This can easily be transformed in the XML format (we use a simple `awk` script to do this):

```
<trace>
  <call entity="" method="send" />
  <call entity="BroadcastEngine" method="marshallAndSend" />
  <call entity="BroadcastEngine" method="broadcast" />
  <call entity="Application" method="launch" />
  <call entity="" method="main" />
</trace>
```

This small trace is represented graphically as a chain graph, shown in Figure 5. Each pending call is represented as a node, and the sequence of nested invocations shown as directed edges. Each call is also labelled with an "observation" time-stamp (here from (0) to (4)) that reflects COSMOPEN's opinion of the time sequence in which the calls occurred (more on this below). In Figure 5 we recognise the interfacing methods `send` and `BroadcastEngine::broadcast`. We also discover a new method, `BroadcastEngine::marshallAndSend`, which is internal to the middleware, and was previously unknown to us.

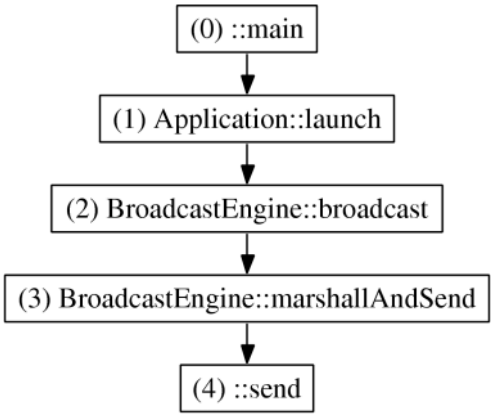


Figure 5: Graphical representation of the first captured trace

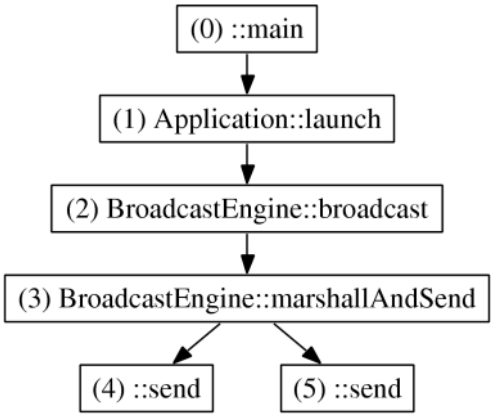


Figure 6: Call graph resulting from trace merging

In most cases, `send` will of course be invoked several times by the middleware, thus yielding more that one trace. Imagine our middleware makes a very economic use of `send` and we only capture a second trace, identical to the previous one. We obtain two identical traces indicating that `send` was called twice. COSMOPEN is able to merge those two traces in one call graph, shown on Figure 6. Note how COSMOPEN assigns timestamp (5) to the new `send` invocation.

Constructing a call graph from stack traces raises a number of issues. Most notably, the reconstructed graph is usually not unique, and multiple solutions can be found for the same set of traces. For instance, the graph of Figure 6 implies that `BroadcastEngine::marshallAndSend` was called only once. There is however no way of asserting this from the two captured traces. We've solved this by choosing to constructs the smallest call graph that is compatible with the observed traces. We'll return to this issue in more details in Section 4, when we discuss COSMOPEN's algorithms.

Enriching observation footprints

The above call graph is very basic. It only captures a small facet of the system's behaviour according to the observation footprint we have chosen. Developers can easily learn more about the middleware by enriching the footprint and thus intercepting a larger set of operations. For instance, if we assume the OS offers a single memory allocation primitive `malloc`, object creations can be monitored by intercepting all invocations to `malloc` at the OS level[2]. In doing so we obtain 4 new stack traces, all ending with `malloc` as their topmost frame. By combining them with the two previous traces (obtained for `send`) we obtain the call graph shown on Figure 7.

This figure illustrates a limitation of call graphs for the representation of the behaviour of large programs. Our example is small in size and only traces two OS system calls (`send` and `malloc`). However the call graph of Figure 7 already contains 14 nodes, and barely fits on the page. Because large and more complex software produces far bigger call graphs (typically with thousands of nodes for the CORBA implementations we consider in Section 5), we have developed a more compact representation, termed *class interaction diagrams*, adapted from

[2] This of course assumes that all object creations cause the OS to allocate memory. This is usually the case for C++. This would not work for instance if the middleware executed on top of a virtual machine with it own memory management. In this case tracing would need to happen directly at the VM API.

the object interaction diagrams commonly found in modelling languages such as UML [OMG99]. This is shown in Figure 8, with COSMOPEN'S graph manipulation tool running in a console, and the current graph under manipulation (corresponding to the call graph of Figure 7) displayed in a viewer window[3].

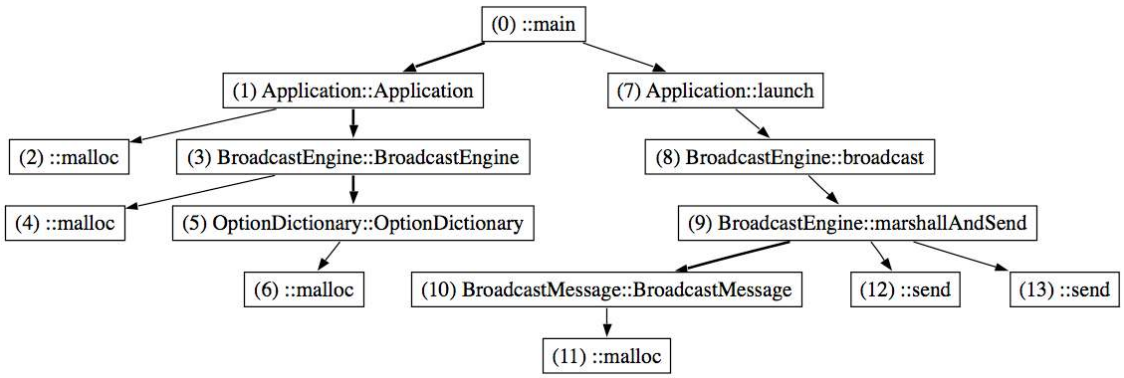


Figure 7: 6 stack traces combined from two OS calls (`send` and `malloc`)

A class interaction diagram collapses onto the same node all invocations performed on the same class. The sequence of invocations can still be followed using sequence numbers. A visual code is also used to distinguish (i) between classes (like `Application` or `BroadcastMessage`, represented in rectangles) and plain functions (`main`, `send`, `malloc`, represented in rounded boxes), and (ii) between plain invocations (thin arrows), and object constructions (thick arrows). The only noticeable distinction between class interaction diagrams and traditional UML interaction diagrams is that individual objects of the same class are not represented separately. As `gdb` provides information about object identity (through the value of the `this` pointer), and COSMOPEN processes this information, implementing plain object diagrams would be straightforward, but would result in more cluttered diagrams, something we were trying to avoid in the first place.

Reading a class interaction diagram as on Figure 8 rapidly becomes natural with practice. More importantly, and as the above example shows, class interaction diagrams are far more compact that standard call trees, which makes then ideal to represent large call graphs. They also allow developers to relate invocations made on the same class that would otherwise appear in completely disconnected parts of a call graph, as for instance in Figure 8 the creation of the `BroadcastEngine` object — invocation n. *(2)* — and the invocation of the method `broadcast` on this object — invocation n. *(7)*.

[3] Our current prototype uses a simple but effective interactive approach: Users can ask COSMOPEN to mirror the graphs being manipulated in an underlying PostScript file (that is the effect of the bind2ps command in the console window). With the right postscript viewer (here ghostview with the "watch file" option on), users can thus visualise the effect of their manipulations as they happen. This feedback loop can be used with both types of representation, traditional call graphs and interaction diagrams.

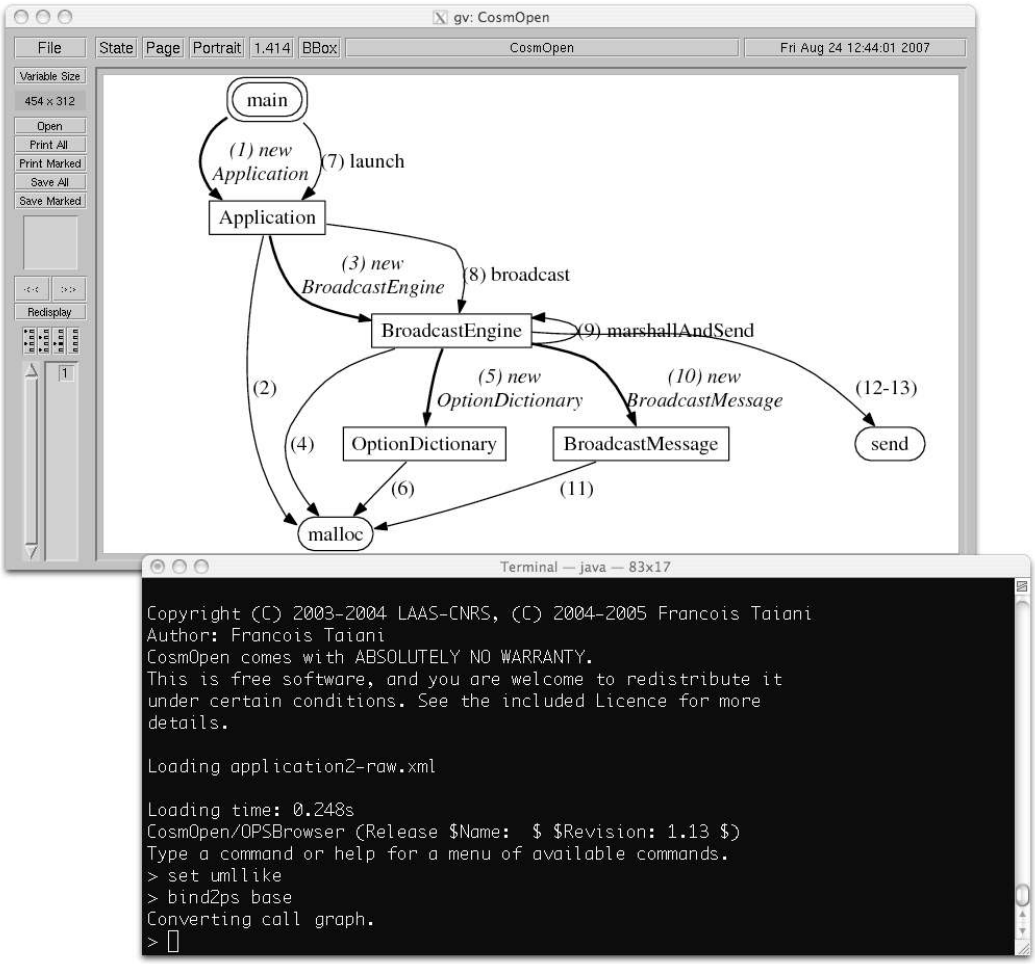


Figure 8: COSMOPEN's graph navigation in action: the call graph of Figure 7 represented as a class interaction diagram

As we have just done with `malloc`, we could further increase the number of traces we collect by simply adding more OS calls to our observation footprint. In doing so however we might collect information irrelevant to our purpose (in the case of our toy example, add fault tolerance to the system), or we might capture in the same graph interactions that belong to different logical planes. In the above example, the calls to `malloc` are in fact part of the C++ implementation of our compiler. We might want to remove them and keep only information about object construction. We might also want to remove the class `OptionDictionary` as it is not primarily related to the handling of broadcast requests. COSMOPEN addresses these issues by providing a graph manipulation language with an associated interactive interpreter. In the next section, we continue our presentation of COSMOPEN by discussing and motivating some of the key capabilities of this engine.

3.3 Untangling Observed Data: A Multi-threaded Example

To illustrate some of COSMOPEN graph manipulation abilities, we now turn to a more complex example, shown in Figure 9. This multi-threaded program creates two POSIX threads, `threadN1` and `threadN2` (using the POSIX function `pthread_create`) and terminates. The two created threads each execute dummy functions (`dummy1` and `dummy2` respectively, not shown) and exit. Our goal here is to observe this program when it runs on Linux (kernel

2.4), and show how COSMOPEN can be used to disentangle the resulting observation data, and separate OS-level mechanisms from the application logic.

```
int main () {
   pthread_t threadN1, threadN2 ;
   pthread_create(&threadN1, NULL, dummy1, NULL) ;
   pthread_create(&threadN2, NULL, dummy2, NULL) ;
   pthread_join(threadN1, NULL) ;
   pthread_join(threadN2, NULL) ;
};
```

Figure 9: **An elementary multi-threaded program**

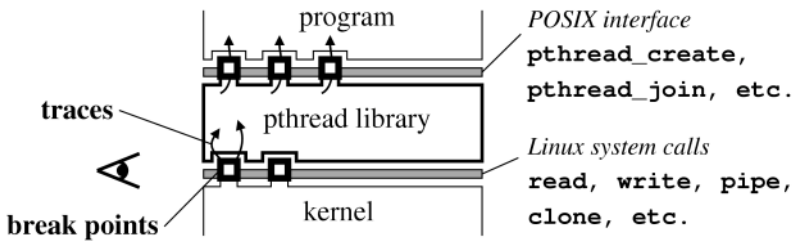


Figure 10: Multithreading Provisioning in GNU/Linux 2.4

On Linux 2.4, the above program executes in a layered environment (Figure 10). It uses a user-space library, called `pthread`, which in turn uses low-level kernel system calls, such as `clone` (creation of a OS-level Linux "process"[4]), `read` (I/O), or `pipe` (IPC) to provide a POSIX-compliant API. This results in a layered architecture, with our small program running on top of the multithreading library, itself running on top of the Linux kernel.

In the following, we show how we can reverse-engineer the program's behaviour even when we limit our observation footprint to the two program-level functions `dummy1` and `dummy2` and a set of low-level OS system calls (`read`, `write`, `pipe`, etc.). For the sake of presentation, we will here assume that we do not know anything of the `pthread` library. As most of the observation footprint is located *below* the library (see Figure 10), our main challenge will therefore be to disentangle the library's execution from the program's own behaviour.

This example is of course oversimplified for the sake of explanation, since we can tell what the program does by simply looking at the code. POSIX is also a well-known standard and we could trace its API directly. This would not be true however for large and complex software, where a manual inspection of source code rapidly becomes unpractical, and intermediary libraries are often little or not documented, as we'll see in Section 5.

3.3.1 Obtaining a first interaction diagram

Applying the approach introduced in the previous section, we apply COSMOPEN to our program and trace a number of kernel system calls (`read`, `write`, `pipe`, `clone`, etc.) along with the functions `dummy1` and `dummy2`. This results in the capture of 14 stack traces, from which COSMOPEN constructs a 28-node call graph, shown in Figure 11 as a call graph, and in Figure

[4] `clone` creates a kernel thread, or light weight process (LWP). In Linux, on most architectures, there is a 1:1 relationship between the light weight processes of the kernel, and the POSIX threads that are visible at the POSIX API. This is not necessarily the case in all operating systems.

12 as an interaction diagram[5]. The diagram in Figure 12 is organised in layers that mirror the structure shown on Figure 10. At the top are application functions (`main`, `dummy1` and `dummy2`); at the bottom kernel system calls (`pipe`, `write`, `read`, `clone`); and in the middle is the multithreading library.

The captured traces belong to multiple threads, some of which are dynamically instantiated. From `gdb`'s output, COSMOPEN can distinguish between each thread and reconstruct how they were created. (We discuss in Section 4 how this is done in more details.) The result of this process is shown in both figures with threads indicated on each invocations (t1, t2, t3 and t4), and thread creation shown as dotted arrows between the creating call and the start function of the new thread. In Figure 12, for instance, arrow number *(6)* between `clone` and `__pthread_manager` means that thread t2 was created when thread t1 invoked `clone` (Linux's system call to create system level threads), and then started executing in `__pthread_manager`. This arrow corresponds to invocation (6) in the raw call graph of Figure 11. On both graphs, we have highlighted the activity of a particular thread, thread t2, whose role we discuss just below.

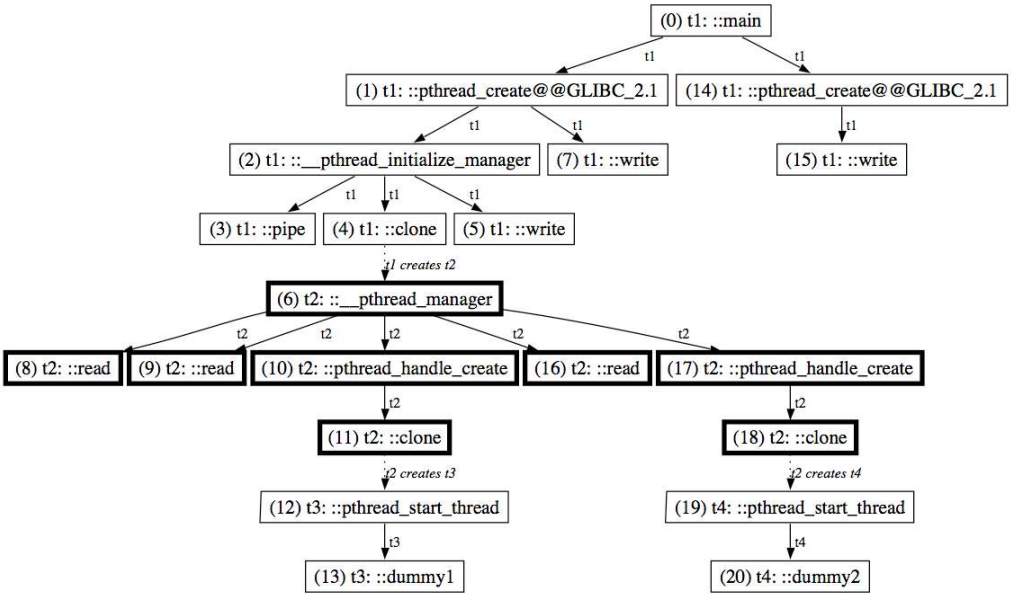


Figure 11: Thread Creation in `libpthread.so` in GNU/Linux 2.4 (raw call graph)

[5] On both figures, two traces related to thread termination have been removed from the diagram for clarity reasons. On Figure 12, the layout of the node with `dot` has been optimised to highlight the different software layers involved.

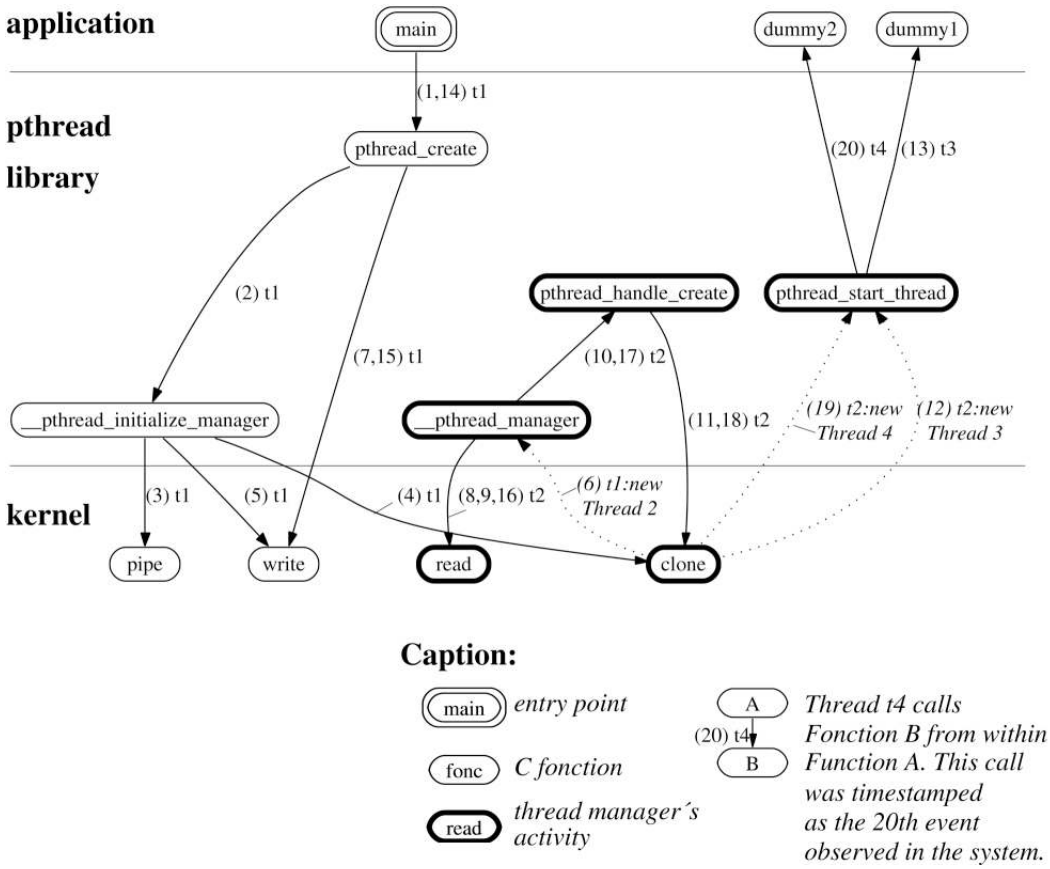


Figure 12: **Thread Creation in `libpthread.so` in GNU/Linux 2.4 (interaction diagram)**

Before we move on to a more in-depth discussion, this diagram raises a number of observations:

- First although the reverse engineered program is very small, the reconstructed call graph is not trivial, and contains quite a few invocations that are internal to the multithreading library (pthread).
- Second, instead of the three threads we could have expected (main thread t1, and the two threads created in the main function), we have here four (t1 to t4): t1 on the left hand side, t2 in bold on the right hand side, and t3 and t4 in the top-right corner.
- Finally the diagram does not reflect the structure of the reverse engineered program: The fact that the main thread t1 creates two threads inside `main` that execute `dummy1`() and `dummy2`() is not apparent.

The fundamental reason for these three points is that by observing the system at the OS API we capture both the behaviour of the multithreading library and of the program we want to reverse engineer. This is a typical case of *cross-layer entangling* that we mentioned in the introduction. The multithreading library is not on the same abstraction level as the application program. It acts as glue between the kernel system calls and the POSIX interface. As a result, the diagram generated by COSMOPEN contains two entangled logical planes (the multithreading implementation and the application logic), yielding a confused picture of the program's actual behaviour.

3.3.2 Multithreading Under Linux: The Raw Call Graph Explained

Before we discuss how COSMOPEN can be applied to this example, we must first briefly explain what is actually happening on the diagram of Figure 12. As in Figure 8, Figure 12 is best read by following invocation sequence numbers. For instance, the first invocation to be recorded (tagged *(1)*) is by thread t1 from `main` to `pthread_create` in the upper left corner. In fact `pthread_create` is called twice by t1, which is the reason why the edge from `main` to `pthread_create` contains two sequence numbers: (1,14), meaning that the second invocation was tagged with the sequence number 14. From `pthread_create`, t1 invokes `__pthread_initialize_manager`. This invocation is the second to be recorded, hence tagged with number (2). t1 then proceeds to invoke three system calls from `__pthread_initialize_manager`: `pipe`, `clone`, and `write`, in this order, tagged with (3), (4), and (5).

With these three calls t1 is actually initialising a special thread called the *thread manager*[6] (numbered t2 on the diagram) that is internal to the library and invisible to external users: (i) with `pipe` t1 initialises a communication channel to communicate with t2; (ii) with `clone` t1 spawns t2; and (iii) with `write` t1 uses the freshly created pipe to send t2 a synchronisation message used for debugging purposes. The creation of thread t2 is represented by a dotted arrow from clone to `__pthread_manager` (an internal library function), and tagged as the 6th observed event.

After executing invocation (5) to `write`, t1 returns from `__pthread_initialize-_manager` into `pthread_create`, and from there sends a thread creation request to t2 on the pipe (invocation (7) to `write`). The thread manager t2 reads the synchronisation message and the thread creation request by invoking the system call `read` twice (sequence numbers (8) and (9)). t2 then creates a POSIX thread t3 that goes on to execute `dummy1` (invocations number 10, 11, 12, and 13). The creation of a second POSIX thread t4 by the main thread t1 proceeds in a similar manner from invocation 14 onward, except that this time the thread manager (t2) does not need to be created.

Admittedly, the above insights into the innards of the multithreading library cannot be inferred solely from the diagram. We had to look up for information on the Web and most importantly study the sources of the multithreading library. The interaction diagram provided however a "core reference map" for this activities, and was invaluable in guiding our analysis. More generally, interaction diagrams provide a crisp representation of the overall mechanisms at play in a program, and help developers know what and where to look for in a code. We get back to this aspect when we discuss case studies in Section 5.

3.3.3 Separating Logical Planes with COSMOPEN

The call diagram shown on Figure 12 contains two overlapping abstraction levels: The low level workings of the `pthread` library, and the higher-level logic of our main program. Although we now understand how the `pthread` library works, we are still limited to a blurred representation of the program's activities. COSMOPEN can be used to sharpen this representation by filtering out the activity of the multithreading library. In doing so, it helps adjusting the "focal length" of the reverse engineering process to a specific abstraction plane. In this case this will reveals the higher application semantics of the program, i.e. the creation of two

[6] The need for an invisible thread manager in the library can be traced back to the design decision made by Linux developers, in particular Linus Torvald, not exactly to follow the POSIX semantics in Linux 2.4 . The reasons for this choice are beyond the scope of this paper but interested readers are referred to [Brown00] for more information.

threads by the program's main thread. In the earth-bound telescope metaphor of Section 2, the ability to discriminate between different abstraction planes is our way to compensate for unobtrusive and cheap observation techniques, and for a lack of knowledge on the internals of a system's components.

In our example, abstracting away the execution of the multithreading library really means abstracting away thread *t*2, the transparent thread manager spawned and used inside the library. *t*2 acts as a proxy for any other threads wishing to perform POSIX based threading operations. Interaction between *t*2 and other threads (in our example *t*1) occurs through a kind of local RPC mechanism based on a pipe IPC primitive. For instance when *t*1 creates *t*3 inside `dummy1`, it first sends a request to *t*2 (invocation 7 to write on figure 11). The actual creation of *t*3, from the point of view of the OS, happens when *t*2 invokes the system call clone (invocation 11 on figure 11) once it has read *t*1's request from the pipe (invocation 9 to read).

In order to abstract *t*2 away, we need to transform the previous sequence of *observed* operations (*t*1 invoking `pthread_create`, *t*1 sending a request to *t*2, *t*2 reading the request, *t*2 creating *t*3) into some higher-level *abstract* event (*t*1 creating *t*3) that reflects the actual semantics of the program rather than low-level implementation details. We want this transformation to be *automatic*, to be able to apply it to potentially very large call graphs.

Our goal, in the remainder of this section, is to show that such a transformation can be achieved by a sequence of relatively simple operations on the underlying call graph, and more importantly that the resulting manipulation are *generic*: They can be applied to any program using the `pthread` library, independently of the program's higher-level logic.

3.4 Lost data dependencies

In our example the programme semantics we wish to represent—the main function creating two threads—is not captured in the call graph by any "graph structure": There is no execution path from the invocation of `pthread_create` by thread t1 to the two `clone` invocations that create the two threads *t*3 and *t*4. The causal link does exists: As explained above, it is because the main thread t1 writes on the pipe shared with t2 that threads t3 and t4 are created. However, since we only trace invocations, this data dependency is lost on us. Because they only represent a program's control-flow, interaction diagrams cannot capture data-induced interactions between threads that may occur through shared structures (data flow). To create an automatic transformation, we need to reconstruct this causal relationship from the clues left in the call graph we have at hand.

Fortunately, we can use here our understanding of the semantics of thread creation in the `pthread` library, and translate that in terms of a simple call graph transformation. The relationship between `pthread_start_thread` and `pthread_create` obey a rather obvious properties that we've termed *pair-wise sequencing*. More precisely, because thread t2 creates new threads in the order in which the corresponding requests are written to the pipe, each call to `pthread_start_thread` can be pair-wise associated with a call to `pthread_create` according to the order in which they occur: The first `pthread_create` with the first `pthread_start_thread`, etc.

Because COSMOPEN tracks temporal time stamps, we can identify for each new thread the invocation to `pthread_create` that created it. Figure 13 illustrates this on our example. It shows exactly the same call graph as Figure 11, except that nodes (e.g. invocations) have been ordered on a timeline according to their sequence numbers: Invocations on the left were recorded by COSMOPEN before those on the right. Four invocations are shown in bold: The two

invocations to `pthread_create` by t1 (invocation number 1 and 14) and, the two corresponding invocations to `pthread_start_thread` (number 12 by t3, and number 19 by t4).

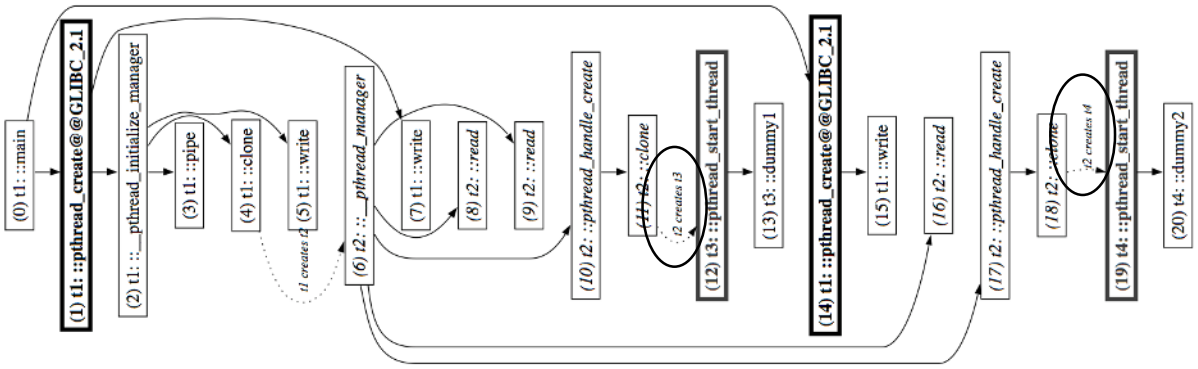


Figure 13: A timeline representation of our program's call graph

Using pair-wise sequencing, we immediately see that invocation n. (1) to `pthread_create` must be associated with invocation (12) to `pthread_start_thread`, and invocation (14) to invocation (19). Once these associations have been found, the call graph can be manipulated to reflect our new understand by changing the parent node of both invocations (12) and (19), as shown on Figure 14: Two new edges now connect invocation (1) with (12), and (14) with (19) respectively. Identifying the sequences of `pthread_create` and `pthread_start_-thread`, and then moving links in the call tree accordingly is an operation directly supported by COSMOPEN (through its `fuse` operator), that can be run on any program using the `pthread` library, as we shall see in Section 4.

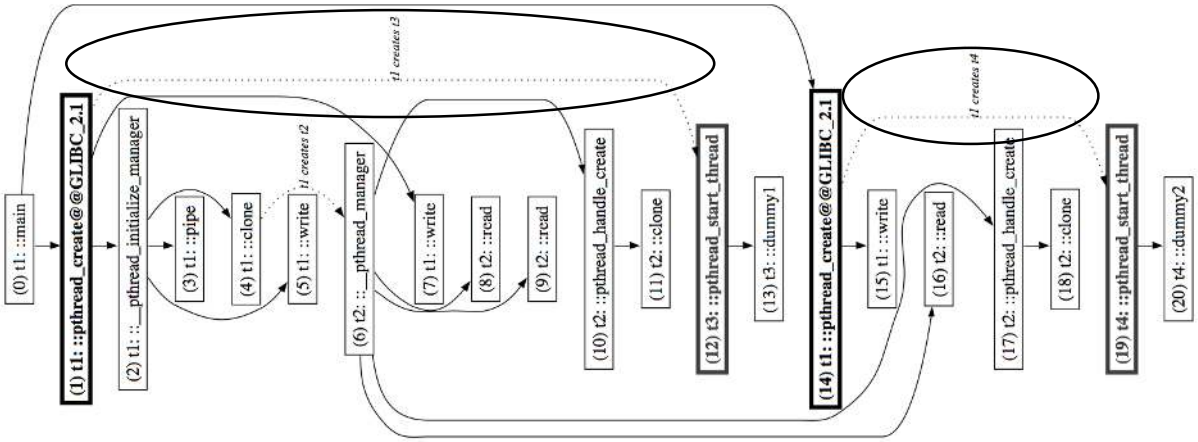


Figure 14: The timeline after the `fuse` operation

There is, however, a difficulty in this process that we've so far silently swept under the rug: In pathological runs, some thread creation might fail. When this happens, COSMOPEN will register an invocation to `pthread_create`, but no corresponding thread start with `pthread_start_thread`, and our pair-wise matching could possibly create erroneous associations. COSMOPEN prevents this aborting the graph transformation is the number of "parent" (here `pthread_create`) and "child" nodes are not equal. By guarding certain graph transformations, we are ensuring that the produced results are always consistent, even though some situations might arise in which a safe transformation is not possible. This conservative approach is closely linked to our philosophy of non-invasiveness and low-cost observation: Be-

cause we have voluntarily limited the data we get access to, cases may in principle arise in which the clues we have gathered are insufficient to accurately reconstruct higher-level models of the programs. This seems a reasonable price to pay, and in the case study we present in Section 5, never actually happened.

The call-graph resulting from this transformation (Figure 14) is shown as an interaction diagram in Figure 16 and as a traditional call tree in Figure 15.

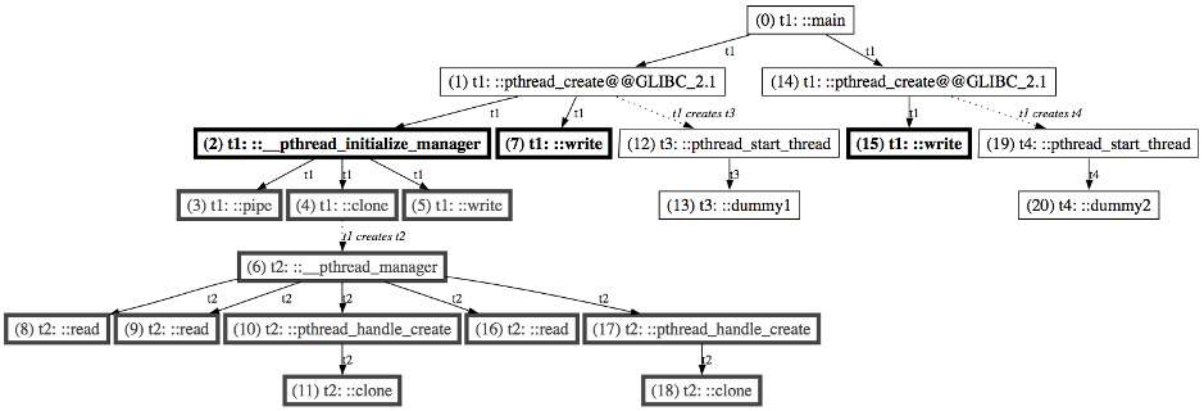


Figure 15: The timeline of Figure 14 represented as a traditional call tree

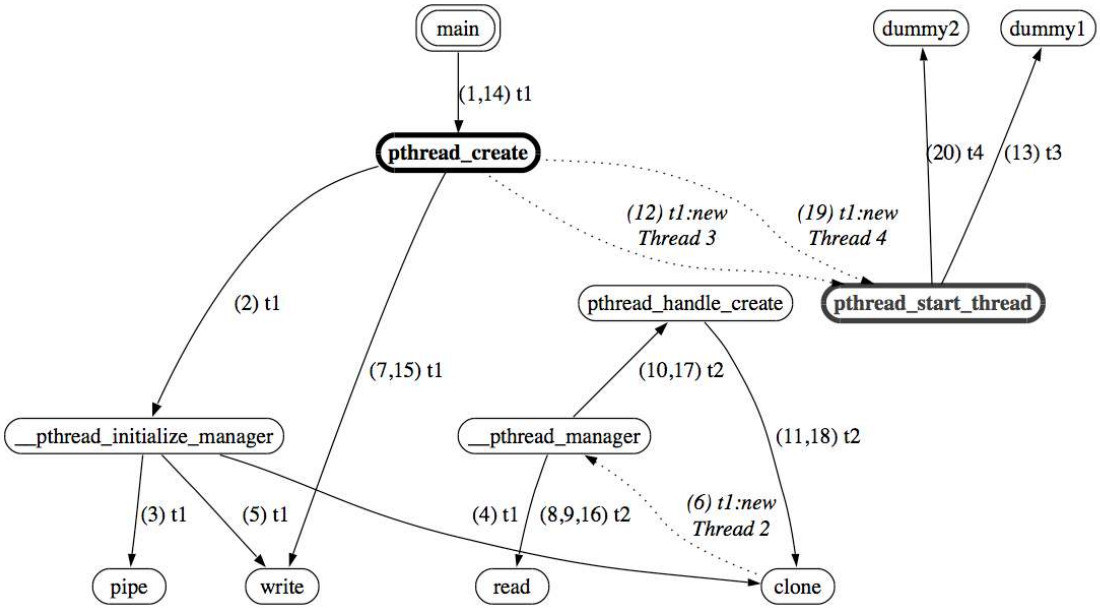


Figure 16: The timeline of Figure 14 represented as an interaction diagram

3.5 Eliminating the threading library

We have progressed in our attempt to reconstruct the original program's logic, but the two above figures still contains quite some superfluous information, in particular pertaining to the inner workings of the `pthread` library. To complete our transformation, we now need to eliminate from Figure 16 all invocations that are internal to the threading library, i.e. all calls represented with a thicker frame on Figure 15.

We could do this by manually removing each individual calls. The resulting graph transformation would however only work on this particular program, and would not serve our goal of generality. Instead we need a way to describe the part of the graph to be removed that will

work even if thread t1 created 10 new threads instead of 2, or if threads t3 and t4 spawned their own sub-threads.

COSMOPEN allows this with operations that combine pattern matching on node names, and navigation of child-parent relationships. In the above example we want to select all children of calls made to `pthread_create`, and then exclude from these children any invocation to `pthread_start`. The resulting set of calls is shown in black bold letters in Figure 15 (invocation 2, 7 and 15). The rest of the sub-graph can then simply be selected by computing the forward closure of these 3 calls. (The closure is shown with bold frames in Figure 15.)

Once selected, these nodes can be removed from the whole graph, resulting in Figure 17.

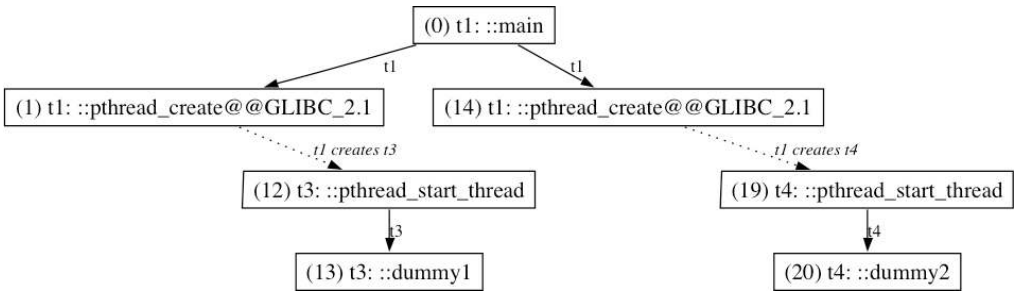


Figure 17: The call tree of Figure 15 once internal pthread calls have been removed

The last steps consists in "abstracting" away the remaining calls belonging to the threading library, i.e. removing calls to `pthread_create` and `pthread_start_thread`, but keeping their children (here dummy1 and dummy2) attached to the call tree. If we do this, we obtained a highly condensed graph, represented as interaction diagram in Figure 18. This last diagram capture the high-level semantic of the small program we started with, and was obtained through a series of simple graph operations on the original call tree that we had reconstructed using a limited observation footprint.

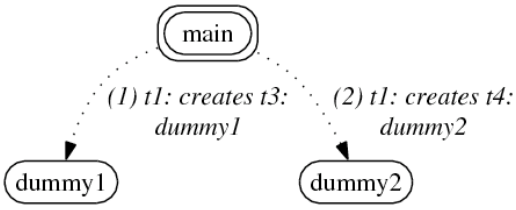


Figure 18: Final interaction graph, after abstracting the remaining POSIX functions

3.6 Conclusion

The actual COSMOPEN commands required to perform the sequence of operations we have just described in shown in Figure 19 below. We discuss the meaning of these commands in more details in Section 4. For the moment, suffice to say that G is the variable that contains the main graph on which we work; CREATE is an intermediate variable that we use to select the internal calls inside the `pthread` library, and that '*' is a wild-card used in pattern-matching expressions.

```
1  fuse    ::pthread_create* ::pthread_start_thread* G

2  put     ::pthread_create* G CREATE
3  forwN   1 CREATE G
4  remove  ::pthread_create* CREATE
5  remove  ::pthread_start_thread* CREATE
6  forward CREATE G
7  exclude CREATE G

8  absPatern ::pthread_create* G
9  absPatern ::pthread_start_thread* G
```

Figure 19: The sequence of commands of COSMOPEN to abstract the pthread library

An essential characteristic of this small manipulation script lies in its generality: Because it uses wildcard expressions and generic graph operations, it's not specifically linked to the small program we've considered, but instead can be re-used as a 'lenses' for the observation of any POSIX program running on Linux 2.4. This generality is a key feature of COSMOPEN: Once created filters can be reused across programs, allowing developers to easily navigate between different level of abstract representations.

4 CosmOpen: reconstructing call-graphs from stack traces

We now discuss in more detail the two main components of COSMOPEN that we've used in the previous: (i) COSMOPEN's dynamic event extractor, and (ii) COSMOPEN's graph manipulation engine. In particular we explain how we reconstruct call-graph from traces, and present the general principles behind COSMOPEN's graph operators.

4.1 Observation on a budget: From stack traces to call graphs

COSMOPEN obtains behavioural information on a program by gathering stack traces on key interface points (the "observation footprint"). This approach works on any platform where calls can dynamically be intercepted and the content of the current thread's stack captured as output. These are standard features available in almost all debuggers, and increasingly found in dedicated interfaces, such as the native JVM Tool Interface (JVMTI) [Sun07], or the Java `java.lang.instrument` package of Java 1.5, both for Java. For C/C++, COSMOPEN currently comes with a standard extractor (`dyngdb`) that relies on gdb for interception and trace capture (Figure 20).

By default, `dyngdb` uses small configuration files (called *gdblets*) that specify which calls to trace, i.e. which observation footprint to use. Internally, `dyngdb` relies on a shared library that encapsulates all low-level details of the interaction with `gdb`. Developers can take advantage of this feature to develop their own customised extractors, as shown in Figure 20. This can be used to vary the scope of the observation footprint while the target program is executing, something we have done in our own case studies (see Section 5). The code inside the callout in Figure 20 illustrates this and shows how a developer would wait for the program being observed to hit the method `foo::bar` before tracing all mutex operations (here specified in a gdblet file).

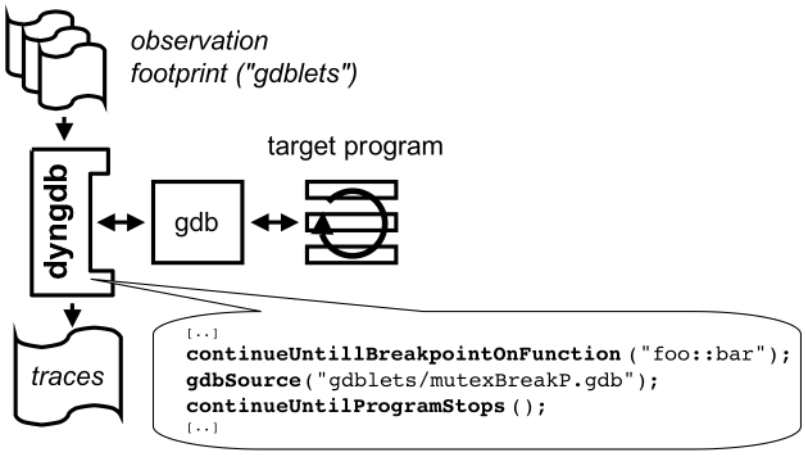


Figure 20: COSMOPEN's Event Extractor for C/C++ (`dyngdb`)

Call-Tree Reconstruction

By collecting stack traces at each breakpoint we obtain information on many more symbols (methods and functions) than the ones where breakpoints are set, and this information can be used to reconstruct a more complete call tree of the program's behaviour. This reconstruction is performed as a front-end operation by COSMOPEN's graph manipulation engine, OPSBROWSER.

Unfortunately, as mentioned earlier, there usually is no unambiguous mapping between a sequence of stack traces and a call tree. If we return to the first two traces we collected in our hypothetical broadcast middleware (Section 3.2), up to 5 call trees can be reconstructed that contain these two traces (Figure 21[7]): In this particular case, the two traces registered by `gdb` are identical (shown on the right of the figure), and the key question is to decide how often each of the involved methods has in fact been invoked.

The first tree (1) can readily be dismissed: Since `gdb` registered two traces, we know at least that the `send` breakpoint was activated twice, and that the `send` function must therefore have been invoked twice as well. We cannot however tell which of the remaining four call-trees is the correct one: On one side, `send` might have been called twice from within `marshallAndSend`, and all other methods called only once (tree (2)); but `main` might as well have called the `launch` method twice, thus triggering twice the same chain of invocation (tree (5)).

This irreducible ambiguity arises because we have limited our observation footprint to a small set of strategic breakpoints. We rely on indirect information—in the form of pending calls—to infer the behaviour of the rest of the program. This approach reduces observation costs, and, as discussed in Section 3, allows developers to easily target aspects of a software they are interested in, but also introduces a part of imprecision. This imprecision is not arbitrary though, and can be dosed to a developer's needs. As a general rule, the more symbols are included in the observation footprint, the less ambiguities will appear in the call graph. An extreme choice would thus be to trace all symbols, which would make the call graph completely unambiguous. Unfortunately, this exhaustive approach quickly becomes intractable on real-life platforms, as we shall see in Section 5. Instead, by allowing developers to calibrate their observation footprint, we allow them to trade off between observation costs and the accuracy of reconstructed models, a key advantage when dealing with large and complex systems.

[7] We have removed all class information, and only kept function or method names for readability's sake.

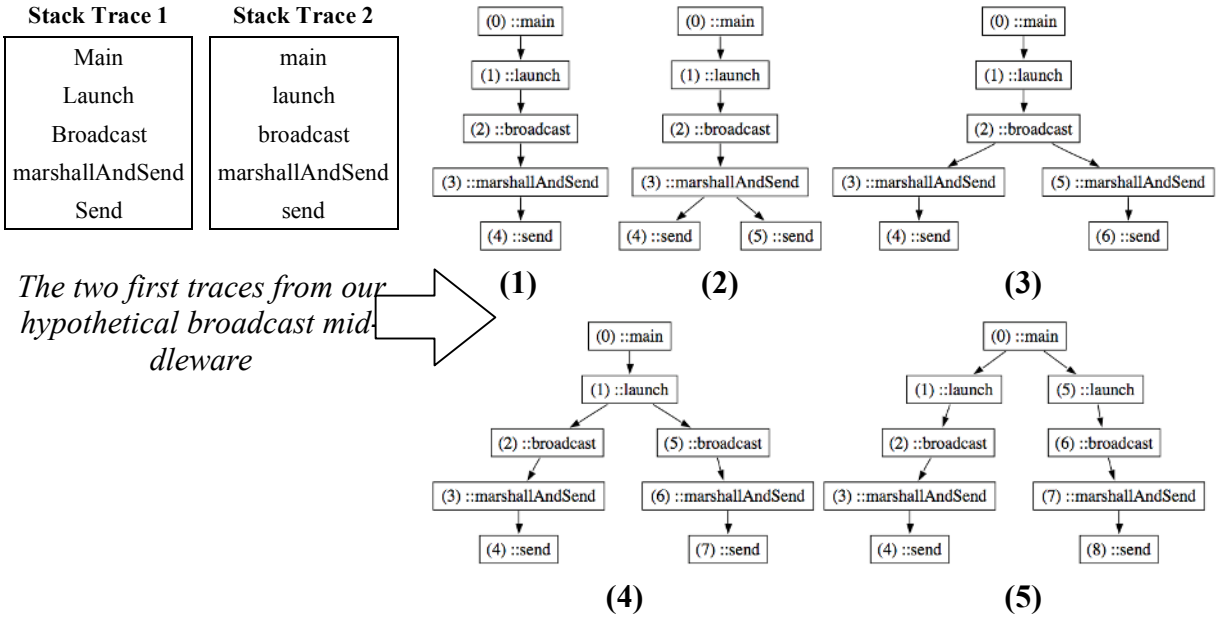


Figure 21: Even the two simple traces of Section 3.2 can produce five different call trees

To resolve this ambiguity, COSMOPEN always opts for *the smallest call-tree compatible with the observed traces*. In the above example, this would be call-tree (2) (as call-tree (1) contradicts the meaning of breakpoints). This reconstruction essentially removes information about possible loops in the program. More precisely, invocations that were looped through multiple times might only appear once in the tree. This choice reflects our two main concerns: (1) we wanted to capture in the tree as much information as could be inferred from the traces; (2) we only wanted to present correct information, in particular regarding ordering of invocations. Choosing the smallest tree means that stack frames get mapped to the same tree node as much as possible, however always under the constrain of the two above rules. From our practice, the resulting trees is in fact quite intuitive, and still reflects the fundamentals of a component's internal behaviour.

This choice of reconstruction also derives from our philosophy of cost limitation: Because we limit ourselves to a partial set of observation points (*the observation footprint*), the raw information we gain is inherently truncated. Our reconstruction algorithm therefore propagates this imperfect knowledge to the call-tree itself, and thus exposes to the developers the inherent cost of information.

The exact algorithm for call-tree reconstruction is sketched in Algorithm 1. Principally, this algorithm works incrementally by adding stack traces to the reconstructed tree in the order in which they were observed. Invocations that are shared with the trace just before (in the algorithm the common prefix between trace^{i} and trace^{i-1}) are discarded (lines 2-6), and the remaining tail of the new trace is hooked up to the existing tree at the point of the last common invocation (lastCommonInvokNode, lines 7-11). Only the very last invocation of the current trace—e.g. `send` in the previous example—is treated differently, and is always added to the tree (line 2). Although this is not shown, the algorithm also keeps trace of the order in which nodes are added (line 9), is the form of logical time-stamps.

```
1   for all trace^i in traceSequence
2       mostOfTheTrace ← trace^i \ [ trace^i.last ]
3       maxPrefix ← longuestCommonPrefix ( trace^(i-1), mostOfTheTrace )
4       lastCommonItem ← maxPrefix.last
5       lastCommonInvokNode ← getInvokNodeForSymbol ( lastCommonItem )
6       newInvokTrace ← trace^i - maxPrefix
7       lastNode ← lastCommonInvokNode
8       for all traceitem in newInvokTrace
9               newInvokNode ← newInvokNodeForTraceitem ( traceitem )
10              createEdge ( lastNode, newInvokNode )
11              lastNode ← newInvokNode
12      end for
13  end for
```

Algorithm 1: Transforming a stack trace sequence into a call tree

Formally, Algorithm 1 can be shown to produce the smallest call-tree that fulfils the following properties[8]:

Property P1: Trace Inclusion

Each observed trace can be mapped to a path in the tree that starts from the tree's root (note that this path may not always end at a leaf node);

Property P2: Breakpoint Discrimination

Each breakpoint activation (e.g. '`send`' in the previous example) is mapped to a node distinct from those of any of the stack frames that precede it.

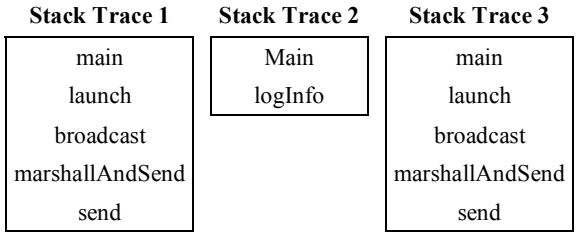
Property P3: Order Conservation

The ordering of siblings in the call tree reflects the order in which traces were observed.

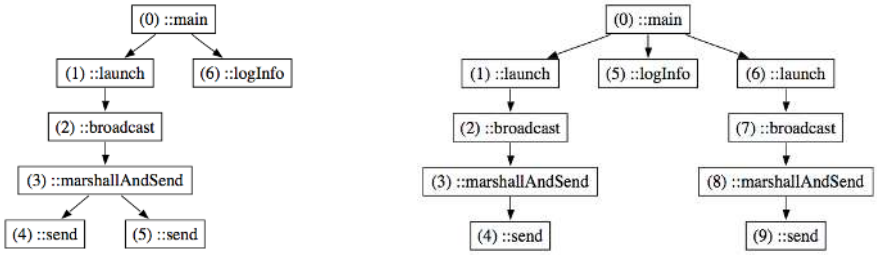
Intuitively, **P1** translates the sequence of nested calls within a stack trace into a path of the call tree; **P2** ensures that we discard Tree (1) in Figure 21 in favour of Tree (2); and **P3** maps the ordering between stack traces into the partial ordering of siblings. For place reasons, we do not give a proof of this result here, but this would be shown rather straightforwardly using a proof by recursion on the number of traces in the trace sequence.

Although Property P3, *Order Conservation*, might seem trivial, it plays in fact a crucial role in ensuring that the correct call tree is being built. For instance in Figure 22, an additional stack trace containing an invocation to the function `logInfo` has been observed between the two traces of Figure 21.

[8] For ease of exposition, we have not included any well-formedness property on the call-tree, e.g. that that the call tree is indeed a tree or that child nodes should be given timestamps higher than their parents.



Three overlapping traces



(a) Smallest tree without Property P3 *(b) Smallest tree with Property P3*

Figure 22: Without Order Conservation Property P2, Tree (a) would be a valid reconstructed call tree for the three traces shown top

Without Property P3, Tree (a), shown on the left, would be the smallest tree respecting Properties P1 and P2: All traces are contained in the tree, and breakpoint activations (here `send`, `logInfo`, and `send`) are pair-wise mapped to distinct nodes. Unfortunately, Tree (a) seems to imply that all `launch` invocations (be it one or more) happened before `logInfo`. It also does not show that `launch` was invoked twice, something we can directly conclude from the traces. It is therefore both incorrect, and incomplete. This happens because Property P3, Order Conservation, is here violated: The second `launch` stack frame of Stack Trace 3 has been mapped to node (1) in the tree, and thus has been ordered *before* the `logInfo` node (6), in contradiction to the fact that Stack Trace 3 was recorded *after* Stack Trace 2. In mathematical jargon, the mapping function between stack frames and tree nodes is here non-monotonic, i.e. it does not conserve ordering.

By contract, Tree (b)—the smallest tree respecting properties P1, P2 and P3—does conserve the ordering of frames in the tree: The two `launch` stack frames (from Trace 1 and Trace 3) get mapped to different tree nodes: one before `logInfo`, and one afterward. Properties P3 is thus respected, and the tree both correctly reflects the ordering of invocations present in the traces, and shows that `launch` was in fact invoked twice.

Multithreading

Multithreaded programs are tackled using an extended version of Algorithm 1. This thread-enabled version uses information about thread creations and thread scheduling provided by `gdb`. In particular, it keeps track of which thread each stack traces belongs to. It also use information related to thread-creation API (the `clone` syscall on Linux) by including it in the observation footprint. Based on this, the algorithm follows a two-stage approach that amounts to constructing individual call-tree for each thread found in the program, and then merging these trees using its knowledge of when the thread-creation API was exercised (thanks to tracing), and when each thread was actually created (thanks to `gdb`). This merge operation is in many ways similar to the `fuse` operator described in Section 3.4. Here however, the tree construction algorithm adopts a more conservative merging protocol to prevent any misinterpretation arising from concurrent thread creation. In particular, the algorithm raises an exception

as soon as multiple thread creations overlap (e.g. a second `clone` is invoked before the thread associated with the first `clone` has been identified).

This safeguard is needed to cater for the broader range of thread semantics in different platforms, and ensures any ambiguous traces are always referred to the developers. Interestingly, we've never encountered such a situation in the examples we have looked at, including the ones described in Section 5, which seems to indicate that except for pathologically high workloads, thread creation latencies are short enough for any ambiguity to disappear.

4.2 A simple yet powerful graph calculus

The call tree constructed by Algorithm 1 often contains several thousands of nodes for non-trivial programs. In Section 3, we've seen how simple graph manipulations could be used to disentangle complex behavioural data. COSMOPEN directly supports these manipulations through a dedicated graph transformation engine that provides an interpreted language optimised to handle the many abstraction planes encountered in complex multi-layer software architectures.

As already hinted at the end of Section 3, one of the key features of COSMOPEN's language is the availability of *graph variables*, that can be used to combine elementary operators in complex filters. Their use is illustrated in the following code excerpt, which loads a graph from the file `jtc.xml` into variable A (`load jtc.xml A`), and the one in `orbacus.xml` into variable `B` (`load orbacus.xml B`). It then assigns the contents of `A` to the variable `C` (`assign A C`), and adds the contents of `B` to `C` (`add B C`)—i.e. `C` ← `A` ; `C` ← `C` ∪ `B`. The resulting graph is then saved in file `orbacus-jtc.xml` (`save C orbacus-jtc.xml`).

```
load jtc.xml     A

load orbacus.xml B

assign A C

add    B C
save   C orbacus-jtc.xml
```

More generally, the commands of COSMOPEN's manipulation language fall into four main categories:

1. **Generic Management Commands** allow users to interact with the interpreter, for instance to set/unset options, run external shell commands, execute a script from a file, or inspect the log of all commands executed so far.
2. **Input/Output Commands** provide support to save and load graphs, and convert graphs both into postscript (with dot's help) and dot format. As explained in Section 3.2, some special 'binding' commands also allow users to constantly reflect the content of a variable into a chosen postscript file, thus providing a simple WYSIWYG mechanism when manipulating graphs.
3. **Transformation Commands** constitutes the biggest group, and provides operators for variable management (assignment, deletion, etc.), set algebra (union, complement), pattern based node selection, and temporal operators, such as the `fuse` operation discussed in Section 3.4.
4. **Extension Commands** provide recursive closure operators to follow invocation chains in graphs. For instance: such operators include a recursive forward closure, which, when applied, includes all the calls directly and indirectly made by the nodes contained in the

input graph. This is this operator that we used to select all the invocation internal to the `pthread` library in Section 3.5.

The main operators for transformation and extension are listed in Table 1 with a short description. With the exception of some temporal commands, they all take a combination of either variables or patterns as parameters. Many operations exist in fact both in a variable-only and a pattern-based form. For instance: '`add A B`' adds the graph content of `A` to `B` (union), while '`put ::pthread_* A B`', adds all the invocation nodes that start with `pthread_` in `A` to `B`. Because node names embed information about classes, methods, thread and time-stamps, this pattern-based selection allows developers to filter invocations according to a wide range of criteria, as we've seen for instance in Section 3.5, when we've removed all invocation of the `pthread` library.

The temporal operators `remAfter`, `remBefore`, and `slice` are slightly different in that they select nodes based on their timestamp, thus allowing access to the temporal information captured by COSMOPEN. '`slice 828-1626 A B`' will add to `B` all invocations from `A` whose timestamps are found between 828 and 1626.

Table 1: The main graph operators of COSMOPEN

Variable management

`clear`	removes all nodes from a graph variable
`delete`	deletes a graph variable
`assign`	assigns the graph of a given variable to a second
`print`	prints the nodes of a graph possibly using a pattern.

Set algebra operators

`add`	adds a graph to another
`exclude`	excludes the nodes of a given graph from a second
`abstract`	abstracts away the nodes present in one graph from another graph
`remAlone`	removes the standalone nodes of a graph

Pattern based operators

`put`	puts nodes from one graph to another w.r.t. a pattern
`remove`	removes the nodes of a graph whose names match a pattern
`absPatern`	abstracts away the nodes that match a pattern
`absSelf`	abstracts away internal calls for classes matching a pattern

Temporal operators

`remAfter`	removes the event of a call graph that are subsequent to a given time
`remBefore`	removes the event of a call graph that are prior to a given time
`slice`	puts nodes from one graph to another w.r.t. a time interval
`leapOver`	leaps over calls based on temporal ordering
`fuse`	recreates child-parent relationships based on temporal sequences
`rebind`	as fuse, but always select the immediately preceding parent

Recursive extension operators

`backward`	computes the backward closure of a graph
`forward`	computes the forward closure of a graph
`spread`	union of the forward and backward closures of a graph
`backN`	computes the backward set of a graph within a given depth
`forwN`	computes the forward set of a graph within a given depth
`spreadN`	computes the spread set of a graph within a given depth
`envlop`	computes the edge envelop of a graph

Methodology

COSMOPEN's operators support a reverse-engineering method made of three fundamental steps: (i) *temporal scoping*, (ii) *seed-based selection*, and (iii) *abstraction*.

The first step (*temporal scoping*) typically uses temporal operators to recover hidden dependencies (such as using a pipe for control flow as in the previous `pthread` example), and scope down the analysis to a particular phase of the program's execution (e.g. after a socket connection has been accepted, but before a CORBA remote method has been executed).

The next step (*seed-based selection*) then consists in selecting a so-called *seed*, a small set of operations one is interested in, and use recursive extension to compute either the calls made by this subset, or the calls leading to this subset. Depending on the complexity of the operation, this seed-extension might be repeated, and the results combined through union and complement.

Eventually the last step (*abstraction*) uses abstraction operators (`abstract`, `absPatern`, `absSelf`) to remove intermediary classes and functions and focus the resulting graph on the key program elements involved in the interaction.

This fundamental pattern is the one followed by the script we presented at the end of Section 3 (repeated in Figure 23 below), which abstracts away the internal behaviour of the `pthread` library from a simple multithreaded program. In line 1, we use the `fuse` temporal operator to recover the thread-creation semantics of `pthread_create` (*temporal scoping*). We then select all `pthread_create` operations as a seed (line 2, *seed-based selection*), and use the depth-bounded forward closure `forwN` (line 3), which in effect select all children of this seed (i.e. all descendant of the nodes in graph `CREATE`, limited to a depth of 1). Lines 4 and 5 further manipulate the closure to keep only those children of `pthread_create` other than `pthread_start_thread` (*abstraction*). The result is used again as a seed in line 6 (*seed-based selection*) to compute all operation internal to the `pthread` library, which are then removed from G (line 7, *abstraction*). The last two lines compact the resulting graph by abstracting away `pthread_` operations (*abstraction*).

```
1  fuse    ::pthread_create* ::pthread_start_thread* G

2  put     ::pthread_create* G CREATE
3  forwN   1 CREATE G
4  remove  ::pthread_create* CREATE
5  remove  ::pthread_start_thread* CREATE
6  forward CREATE G
7  exclude CREATE G

8  absPatern ::pthread_create* G
9  absPatern ::pthread_start_thread* G
```

Figure 23: The script we used to abstract the pthread library in Section 3

4.3 Conclusion

This ends our overview of COSMOPEN's event extractor, call-tree reconstruction, and graph manipulation engine. COSMOPEN's whole design of follows our objective of adjustable observation costs: Our prototype does not require any particular instrumentation, except for a debugging-enabled version of an application, and only requires a subset of invocations to be intercepted, depending on which aspects a developer wishes to investigate (lock synchronisation, life-cycle of requests).

Observation policies can be tailored to the specificities of an application, a particularly interesting ability for complex long-running systems, where only one particular phase of the platform's lifecycle might be of interest. The accompanying graph operations provided by our tool are simple yet powerful, thanks to an interpreted script language featuring graph variables to store intermediary results. This allow for reusable abstraction scripts to be created, as we have shown at the end of section 3.

In terms of method, and although we allow for any strategy, including incremental approaches, the standard tactic we have developed is to intercept every call on the lower and higher interfaces of a component (e.g. for a request-reply middleware, all syscalls and all remote services implemented at the application level), and use graph manipulation according to a three-step approach (*temporal scoping, seed-based selection* and *abstraction*) to create higher-level representations.

5 Case study: non-determinism in a CORBA ORB

In this last main section, we present how we have applied COSMOPEN to three well-known industrial CORBA platforms (ORBACUS [ORBacus04], TAO [Schmidt99], and OMNIORB [Grisby04]) to analyse how multithreading could cause replicated requests to be delivered inconsistently across replicated servers. We have presented elsewhere [Taïani03, Taïani05] the results of our findings (for ORBACUS). In this section, we report on how we used COSMOPEN to conduct this analysis, with a particular focus on ORBACUS.

We first briefly introduce CORBA, and present the context of our work in more detail. We then describe how we recorded execution traces from the different middleware we considered and explain how we used COSMOPEN to extract a high level behavioural pattern from our raw observation data.

5.1 CORBA and Fault-Tolerance

CORBA is a standard for communication-oriented middleware developed by the OMG [Corba02]. It defines a norm for *Object Request Brokers* (ORB), based on the notion of *distributed objects*, and *remote method invocation*. It compares to other ORB standards, such as Java RMI or DCOM. CORBA's main strength lies in its independence from any language and any platforms. With CORBA, programs developed in different languages (Java, C++), running on different OS (Windows, Linux, Solaris *etc.*), can be easily "glued" together to build complex distributed applications (Figure 24).

Over the years, CORBA has come to integrate many additional technologies, such as distributed events, transactions, real-time capabilities, or component based development. An important effort was made in particular to include fault-tolerance into CORBA, and resulted in the specification of Fault-Tolerant CORBA (FT-CORBA) [FTCorba02]. FT-CORBA does not cover all needs of distributed fault-tolerant application, and in particular does not provide any means to control non-determinism in multi-threaded middleware. To address this issue, we decided to use COSMOPEN to identify where different middleware implementations could

cause an application to behave in a non-deterministic manner. Among others, we needed to know when lock allocation decision made by the kernel could influence the internal request handling of the middleware.

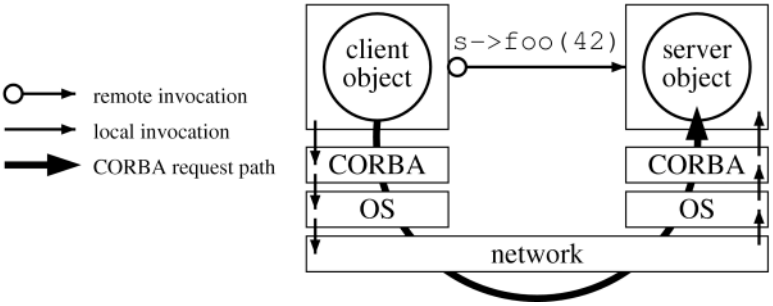


Figure 24: **Remote Method Invocation in CORBA**

This required that we understood precisely how requests progress from the network interface (network related OS syscalls such as `socket`, `recv`, `send`, etc.) to the application (application level interface), while tracking synchronisation activities (essentially mutex operations) along the way. All three platforms being implemented in C++, we also wanted to trace object creations and deletions in the form of memory allocations (`malloc`, `free`, etc.).

5.2 *In vivo* observation of multithreaded industrial middleware

To observe request processing in the three ORBs in vivo, we wrote up a basic CORBA object that prints a "Hello World" message on the console, and traced the activity of the server while one request was processed and the server subsequently terminated. Following the approach that we presented in Section 3.2, we identified a set of 59 breakpoints on the Linux kernel and its associated system libraries that we needed to track to perform the above analysis. 18 breakpoints were related to low level IPC (`pipe`, `accept`, `select`, `read`, `write` *etc.*), memory management (`malloc`, `free`, `calloc`, *etc.*), and process management (`clone`, `wait4`). 23 breakpoints monitored the activity of the multithreading library (`libpthread.so`). 18 breakpoints were set to observe lock activity that did not use the `pthread` API (`mlock`, `flock`, *etc.*). We also set 3 additional breakpoints at the application level to observe upcall CORBA invocations.

Tracking memory allocation, mutex activity and input/output considerably slows down a program's execution, most particularly during its initialisation phase, as shared libraries are loaded, initial objects are created, and configuration data is retrieved. With all breakpoints set, observing a single ping-pong request on ORBACUS on a 1GHz Pentium III server running Linux kernel 2.4 took more than one hour (1h 2min 11s precisely). 99.2% (1h 1min 45s) of this time was passed initialising the ORB, which is not the phase we were interested in, as no request is processed at this time. For comparison, a non-monitored run would take less than 1 second on the same hardware. To avoid this intractable observation cost, we applied the adaptive observation approach described in section 4.1, and automatically activated the most costly breakpoints only after the middleware was initialised. This reduced the duration of a monitored run with ORBACUS to 4min 53s (283s), only 8.8% of which was spent in the initialisation. This more than tenfold increase in performance shows the substantial gains a customised observation policy can bring in complex platforms.

The number of threads, traces, items and invocations obtained for TAO and OMNIORB are shown on Table 2. For instance on ORBACUS, we collected 658 stack traces spanning the behaviour of 8 different threads and totalling 9178 stack frames. Based on these, the tree-

reconstruction algorithm we presented in section 4.1 produced a call tree containing 2066 invocation nodes.

One rationale behind the use of breakpoints to generate a call tree is that a single breakpoint activation actually yields information about all the symbols present in the active thread's stack, not only about the breakpoint definition point. As a measurement of the efficiency of the event collection we computed the ratio between the size of the final call tree (which represents the meaningful information we can analyse) and the number of stack traces (which each correspond to a breakpoint activation, and hence to an observation cost) for each ORB. As the table shows, there is an important disparity between the ratios (3.13 for ORBACUS, 1.68 for OMNIORB). These disparities come from the different design choices found in the ORBs. OMNIORB often uses polling loops controlled by timers for different mechanisms. In particular it launches a scavenger thread that is regularly activated for garbage collection. Because our tree-building algorithm tends to collapse loops into one instance, those loops generated many traces (in OMNIORB the scavenger thread produced 890 traces, i.e. 48% of the total number) that do no yield many new invocation nodes.

Table 2: Size of observation data for one ping-pong request

ORB	threads	traces	frames	invocations	invocations/traces
ORBACUS 4.1	8	658	9178	2066	3.13
OMNIORB 4	7	1828	16807	3088	1.68
TAO 1.2.1	6	512	11260	1352	2.64

The breakdown of the breakpoint activations leading to these traces is represented on Figure 25 for the three ORBs. We notice that most of the collected traces (82%) are related to mutex synchronisation. In ORBACUS for instance, 538 mutex operations were observed. Only a subset of these operations is however directly related to request processing: Still in ORBACUS for instance, only 203 out of 538 mutex operation occur between request reception (return from the OS call `recv`) and the corresponding reply being send (return from the OS call `send`). This second figure is more representative, as it does not take into account either bookkeeping operations, or binding overheads (i.e. the set-up of a socket-oriented connection between client and server objects), and correspond to the critical path of a request within the ORB. Table 3 compares the number of mutex operations between reception and reply of a request in ORBACUS, OMNIORB and TAO. We see that the synchronisation activity of ORBACUS is significantly higher than in others ORBs, but, and this may come as a surprise, OMNIORB and TAO do indeed generate an important number of mutex operation themselves.

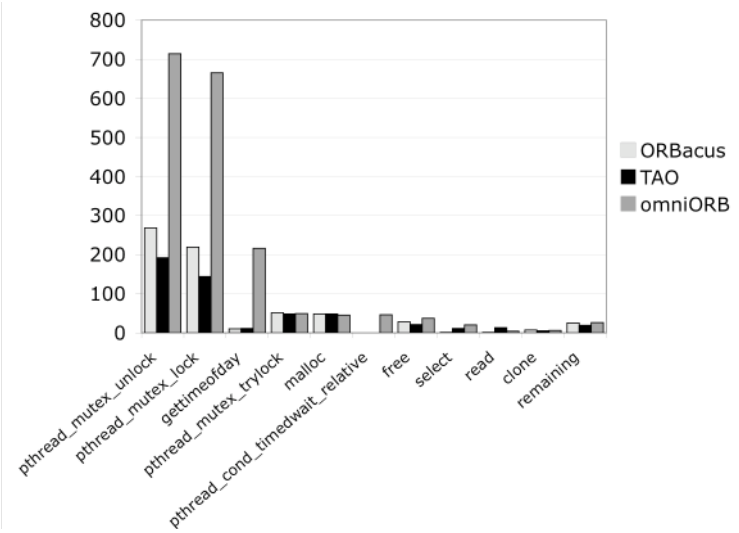


Figure 25: Breakdown of activated breakpoints in ORBACUS, TAO, and OMNIORB

Table 3: Lock operations during request handling in popular ORBs

ORB	Lock operations
ORBACUS 4.1	203
OMNIORB 4	64
TAO 1.2.1	52

To understand the possible sources of non-determinism, we needed to grasp the role played by the mutex operations of Table 3 during the processing of our request, and to do this we first needed to understand what lifecycle requests would go through in each ORB. In the remainder of this section, we present how we used COSMOPEN to this aim.

5.3 Extracting high level behavioural patterns with OPSBROWSER

Directly analysing any of the obtained call trees is in practice intractable, due to their size. Figure 26 shows for instance the complete interaction diagram obtained for ORBACUS. This graph totalises more than 2000 individual invocations, over 50 C-functions and 140 C++ classes. Its sheer size makes it extremely hard to read, to say nothing of any manual analysis. We explain here how we have used COSMOPEN's graph transformation engine to navigate between the different abstraction planes contained in this graph, and extract crisp and meaningful behavioural patterns.

The general approach we have used to analyse the above graph and navigate between its various abstraction planes is the same as for the smaller examples we presented in Section 3. We repeatly applied the three steps of *temporal scoping*, *seed-based selection*, and *abstraction* that we introduced in the previous section. Each incremental steps gave us new insights in the structure and behaviour of ORBACUS, and helped decide which actions should follow.

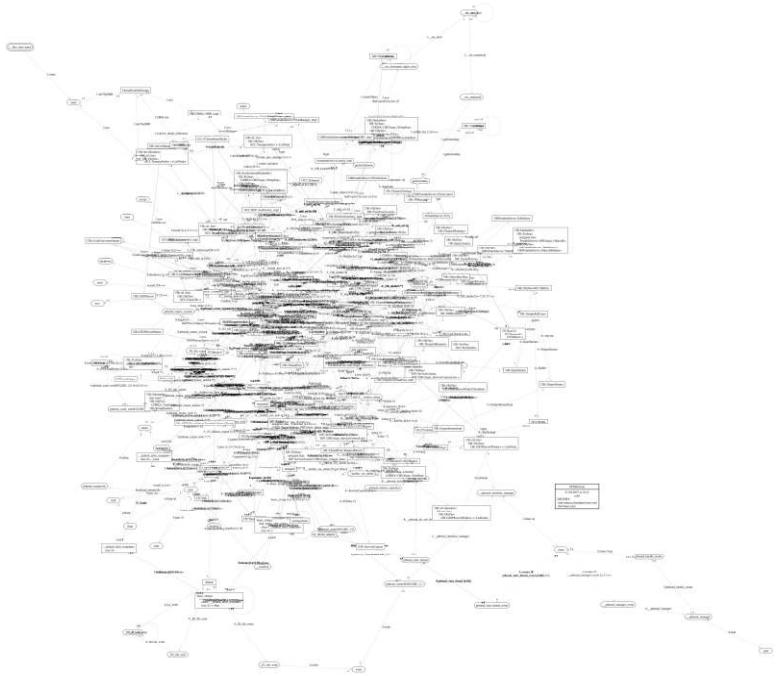


Figure 26: The complete call-tree or ORBACUS reconstructed by COSMOPEN

In terms of scoping, we first started by abstracting away the multithreading library, as explained in Section 3.3. This only removed 30 invocations from the original graph, but most importantly, it created the proper thread creation links. We then searched for networking activity in our graph (using COSMOPEN's `print` command), and discovered that `recv`, and `send` were the only primitives used for network communication by ORBACUS. (Note that different ORB might use different primitives. TAO for instance uses `read`, and `writev`.) We used these network invocations, along with the application activation point (a dummy `Hello_impl::say_hello` method in our case), as our *seed*, and used recursive extension to get a fuller pictures of all invocations in the ORB that would resulted either in request reception (`recv`), reply (`send`), or application invocation (`say_hello`). The corresponding COSMOPEN script is shown below (`GlobalGraph` is the complete graph of Figure 26, `R` is the result graph) .

```
put ::recv'* GlobalGraph R
put ::send'* GlobalGraph R
put Hello_impl::say_hello'* GlobalGraph R
put ::accept'* GlobalGraph R
backward R GlobalGraph
```

This produced a graph containing 52 invocations which made apparent some of the internal libraries used by ORBACUS for communication and multithreading. Using the same approach as for the `libpthread.so` system library, we removed these libraries from the call graph (*abstraction*), ending with 27 remaining invocations. Relying on this graph as a backbone to inspect ORBACUS' code, and constructing complementary graphs to analyse mutex usage and the activity of various other threads, we identified additional key entities or activities (such as the instantiation of the class `Upcall`, and the `add` and `get` operations on the `ThreadPool` class), that we used as seeds and then recursively expanded before adding them to our current

graph. (In the code below, `U` and `TP` are the seeds that are expanded, `R` is the result graph to which `U` and `TP` are added.)

```
put OB::Upcall::Upcall'* U
backward U GlobalGraph
add U R
put OB::ThreadPool::add'* TP
put OB::ThreadPool::get'* TP
backward TP GlobalGraph
add TP R
```

The resulting graph contained 36 invocations. The final step to attain a more compact representation was to remove classes that only acted as delegates, leading us to the final graph, shown in Figure 27. This diagram totals 27 invocations, to be compared to the 2066 ones of the original graph. It clearly shows the actions taken by four different threads ($t1$, the main thread, $t3$ the thread that accepts connections, $t8$ the receiver thread, and $t4$ the working thread) at initialisation (Steps (1) to (8)) and during the processing of a first request (Steps (9) to (26)). The graph covers thread creations ($t3$ is spawned by $t1$ in Step (5), $t8$ by $t3$ in Step (10) for instance) and object allocations (a new `ThreadPool` object is allocated by Thread $t1$ in Step (4)). For clarity reasons, mutex activity is not represented on this figure, but if it were, it would show that (8) `t4:get` and (17) `t8:add` use a mutex to coordinate their access to the `ThreadPool` object. This is typically the kind of operation we were interested in our analysis, to finely control the determinism of the middleware execution.

This example shows on an industry-grade software how the capabilities of COSMOPEN can be used to extract from an entangled graph of raw behavioural data (Figure 26) a higher-level representation highlighting some key aspects of the platform (here request processing, shown Figure 27), and this for a very reasonable cost of observation.

6 Related work

We've already discussed existing tools in our problem statement section (Section 2). Here we briefly revisit some of them in the light of what we have presented, and extend our comparison to areas beyond pure reverse-engineering.

As mentioned in our problem statement, early reverse-engineering work was essentially focussed on static structures and source code analysis. The past decade has however seen numerous efforts to better integrate dynamic data in the reverse engineering process [Reiss-03, Ebert02, Jerding97, Pauw00, Reiss00, Richer-99, Stroulia-02,Wong95]. Among these efforts, one of the closest works to our own is probably the work of [Richer-99] which proposes a domain-specific query language to extract higher-level models from both static and dynamic information. This language, based on Prolog, is focussed on the structural notion of "components" obtained through clustering of program entities, something we do not consider here. Instead, we've primarily looked at the tension between observation costs and information completeness, and at issues of cross-layer entangling in layered platforms, two topics that to the best of our knowledge have rarely been addressed in the literature.

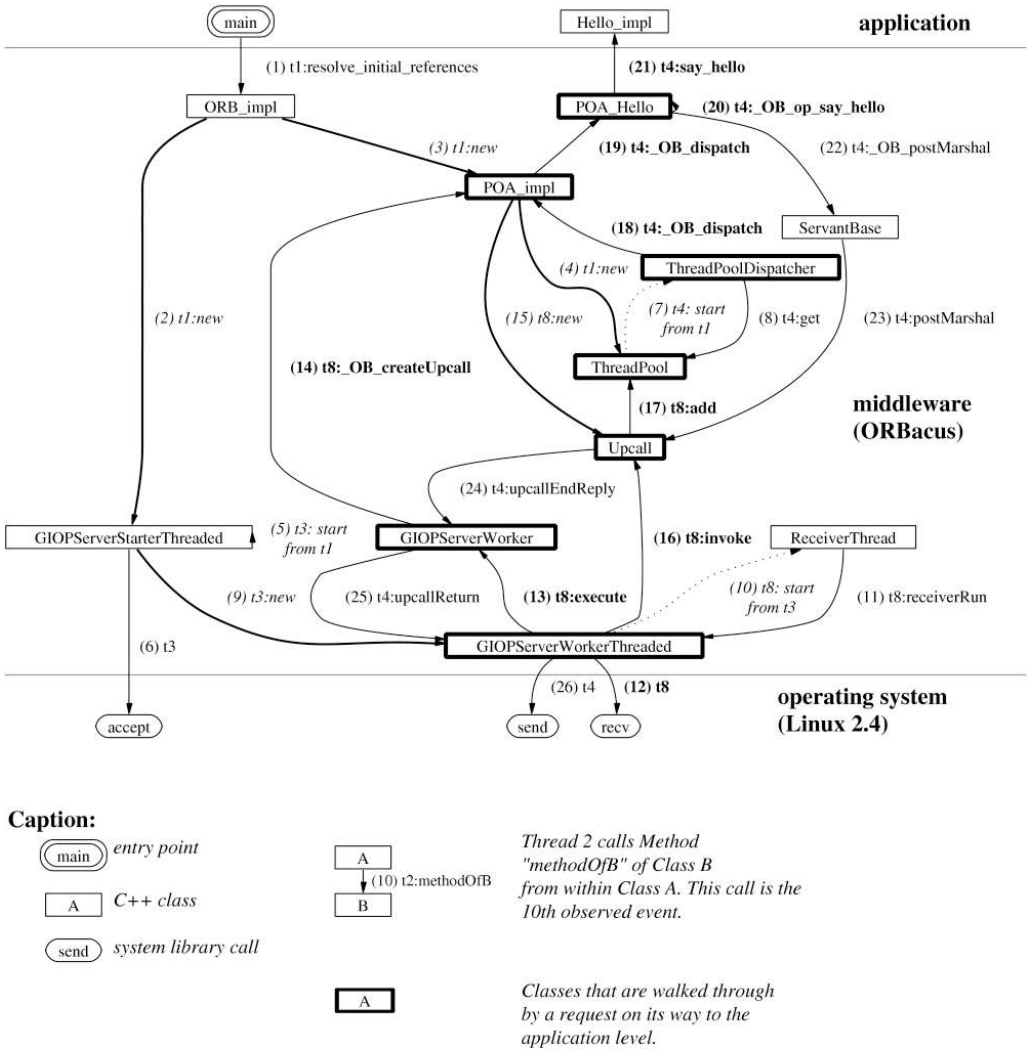


Figure 27: High level representation of the request processing in ORBACUS

Interestingly, we think our work is also related to efforts in other fields that strive to integrate information from different level of abstraction into one consistent representation, such as in distributed debugging [Ottogalli01], or vertical profiling [Hauswirth04]. We also see a strong link to work that reify hidden structures from code artefacts, in particular in the active area of aspect mining (see for instance [Shepherd06, Coelho06]). These work differ from ours in that they focus on cross-cutting concerns, while our main concern has been to address cross-layer entanglement, while maintaining flexible and low observation costs.

7 Conclusion and perspectives

We have presented COSMOPEN, a powerful and practice-driven tool for the behavioural observation and analysis of complex multi-layer platforms. Building on the lessons learnt from former reverse engineering tools, COSMOPEN combines a cheap, and non-intrusive approach for dynamic observation of programs with a simple yet powerful interactive graph calculus to help navigate through the logical planes founds in modern systems. Thanks to its inexpensive event extraction scheme, COSMOPEN can be applied to large industrial software without instrumentation, as we have shown with our case study of popular CORBA middleware. Its graph calculus engine, OPSBROWSER, implements a novel approach to the navigation and

visualisation of large behavioural data, based on partial collapsing. With OPSBROWSER, crisp and meaningful behavioural patterns can be extracted from the execution traces generated by a program run, and can be used to drive the understanding of a component's internal behaviour. Most notably, OPSBROWSER specifically targets multi-level software, by allowing its user to adjust the "focal length" of the reverse-engineering process to specific abstraction planes.

COSMOPEN and its abstraction component OPSBROWSER still suffer from numerous limitations that partly arise from the deliberate choice of a simple, cheap and non-intrusive observation approach. Potential further developments include the integration of OPSBROWSER into an integrated development environment (IDE) such as Eclipse [Eclipse07], to add code browsing facilities, and graphical interaction mechanisms. Another improvement we are considering relates to the explicit management of abstraction planes, to prevent inconsistent graph manipulations.

As open-source software gains industrial relevance, we think the COSMOPEN suite represents a useful practical step to help analyse inter-component interactions in complex multi-layer software platforms. We expect that sort of approach to be of great use to organisations that need an in-depth understanding of the third party components they employ and help them harness the power of available open-source components.

8 Bibliography

[Chen95] Y.-F.R. Chen, G.S. Fowler, E. Koutsofios, and R.S. Wallach. Ciao: a graphical navigator for software and document repositories. In International Conference on Software Maintenance, pages 66-75, Opio (Nice), France, 1995.

[Chen97] Yih-Farn R. Chen, Emden R. Gansner, and Eleftherios Koutsofios. A c++ data model supporting reachability analysis and dead code detection. SIGSOFT Softw. Eng. Notes, 22(6):414-431, 1997.

[Corba02] Common Object Request Broker Architecture: Core Specification (Version 3.0.2 - formal/02-12-02). Needham, MA, U.S.A., December 2002.

[Ebert02] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO. Generic Understanding of Programs - an overview. Electronic Notes in Theoretical Computer Science, 72(2), 2002.

[Eclipse07] Eclipse platform technical overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf, January 2007

[Emden00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. Software - Practice and Experience (SPE), 30(11):1203-1233, 2000.

[Frohlich94] M. Frohlich and M. Werner. Demonstration of the interactive graph visualization system davinci. In Proceedings of DIMACS Workshop on Graph Drawing'94, volume 894 of LNCS, pages 266-269. Springer-Verlag, 1994.

[FTCorba02] Common Object Request Broker Architecture: Core Specification (Version 3.0.2 - formal/02-12-02), chapter 23. Fault Tolerant CORBA. In [OMG99], December 2002.

[Graham83] S. Graham, P. Kessler, and M. McKusick. Execution profiler for modular programs. Software - Practice and Experience, 13:671-685, 1983.

[Graphviz07] Graphviz - Graph Visualization Software, http://www.research.att.com/sw/tools/graphviz/, accessed August 16, 2007

[Grisby04] Duncan Grisby, Sai-Lai Lo, and David Riddoch. The omniorb version 4.0 user's guide. http://omniorb.sourceforge.net/omni40/omniORB/, July 2004.

[Jerding97] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In Proceedings of the 19th international conference on Software engineering, pages 360-370. ACM Press, 1997.

[Killijian00] Marc-Olivier Killijian and Jean-Charles Fabre. Implementing a reflective fault-tolerance CORBA system. In 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000), pages 154-163, Nürnberg, Germany, 2000.

[Mitre03] Use of free and open-source software (FOSS) in the u.s. department of defense. Technical Report MP 02 W0000101 (Version 1.2.04), The MITRE Corporation, January 2003.

[O'Byrne96] John O'Byrne. Sharper eyes on the sky. Sky & Space Magazine, pages 20-24, December 1996.

[OMG99] OMG unified modeling language v. 1.3. http://www.omg.org/cgi-bin/doc?ad/99-06-08.pdf, June 1999.

[ORBacus04] Orbacus user's guide v.4.2.0. http://www.orbacus.com/support/new_site/manual/4.2.0/users_guide/ob.pdf, June 2004.

[Ottogalli01] F.-G. Ottogalli, C. Labbé, V. Olive, B. de Oliveira Stein, J. Chassin de Kergommeaux, and J.-M. Vincent. Visualisation of distributed applications for performance debugging. In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C.J. Kenneth Tan, editors, ICCS'01: International Conference in Computational Science, Lecture Notes in Computer Science 2074, pages 831-840, Berlin, Heidelberg, 2001. Springer.

[Pauw00] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. Concurrency: Practice and Experience, 12(14):1431-1454, December 2000.

[Pérennou98] Tanguy Pérennou and Jean-Charles Fabre. A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach. IEEE Trans. on Computer, Special Issue on Dependability of Computing Systems, 47:78-95, 1998.

[Reiss00] Steven P. Reiss and Manos Renieris. Generating java trace data. In Java Grande Conference (ACM 2000 conference on Java Grande), pages 71-77, San Francisco, CA, USA, 2000. ACM.

[Reiss-03] Steven P. Reiss, Manos Renieris, The BLOOM Software Visualization System, in Software Visualization: from Theory to Practice, K. Zhang, Ed. Kluwer Academic Publishers, 2003

[Richner-99] Tamar Richner and Stéphane Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In Proceedings of 7th International Conference on Software Maintenance (ICSM'99), pages 13--22, August 1999

[Schmidt99] Douglas Schmidt and Chris Cleeland. Applying patterns to develop extensible orb middleware. IEEE Communications Magazine, 16(4), April 1999. Special Issue on Design Patterns.

[Stallman02] Richard Stallman and Roland H. Pesch. Debugging with GDB. The Free Software Foundation, Boston, MA, USA, 9th edition, 2002.

[Stolper99] Steven A. Stolper. Streamlined design approach lands mars pathfinder. IEEE Software, 16(5 (September/October)):52-62, 1999.

[Stroulia-02] Eleni Stroulia and Tarja Systä, "Dynamic Analysis For Reverse Engineering and Program Understanding", Applied Computing Review, ACM, vol 10, issue 1, 2002.

[Taïani03] François Taïani, Jean-Charles Fabre, and Marc-Olivier Killijian. Towards implementing multi-layer reflection for fault-tolerance. In The International Conference on Dependable Systems and Networks (DSN-2003), San Francisco, CA, June 2003. IEEE Computer Society.

[Taïani05] François Taïani, Jean-Charles Fabre, Marc-Olivier Killijian , A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures, The International Conference on Dependable Systems and Networks (DSN'2005), Yokohama, Japan, June 28 - July 1, 2005, pp.270-279

[Wong95] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. IEEE Software, pages 46-54, January 1995.

[Wong98] K. Wong. The Rigi User's Manual - Version 5.4.4. Department of Computer Science / University of Victoria, Victoria, BC, Canada, June 30 1998.

[Hauswirth04] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, Michael Hind, Vertical Profiling: Understanding the Behavior of Object-Oriented Applications, 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2004

[Shepherd06] Shepherd, D., Pollock, L., and Vijay-Shanker, K. 2006. Towards supporting on-demand virtual remodularization using program graphs. In Proceedings of the 5th international Conference on Aspect-Oriented Software Development (Bonn, Germany, March 20 - 24, 2006). AOSD '06

[Coelho06] Coelho, W. and Murphy, G. C. 2006. Presenting crosscutting structure with active models. In Proceedings of the 5th international Conference on Aspect-Oriented Software Development (Bonn, Germany, March 20 - 24, 2006). AOSD '06. ACM, New York, NY, 158-168

[Brown00] Zack Brown, Posix Threads (pthreads) In Linux, Kernel Traffic #84 For 11 Sep 2000 , http://www.kerneltraffic.org/kernel-traffic/kt20000911_84.html

[Tatsubori00] Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian and Kozo Itano, OpenJava: A Class-Based Macro System for Java, Lecture Notes in Computer Science 1826, Reflection and Software Engineering, , Walter Cazzola, Robert J. Stroud, Francesco Tisato (Eds.), Springer-Verlag, pp.117-133, 2000.

[Chiba03] Shigeru Chiba and Muga Nishizawa, An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.364-376, Springer-Verlag, 2003.

[Schwarz02] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of Executable Code Re-visited. Proc. 2002 IEEE Working Conference on Reverse Engineering (WCRE 2002), Oct. 2002.

[Sun07] Sun Microsystems JVMTM Tool Interface, Version 1.0, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, accessed Jan. 2007