COSMOPEN: Dynamic reverse engineering on a budget

How cheap observation techniques can be used to reconstruct complex multi-level behaviour

François Taïani[1], **Marc-Olivier Killijian**[2], **Jean-Charles Fabre**[2,3]
[1] **Computing Department, Lancaster University, Lancaster, UK**
[2] **LAAS-CNRS, Université de Toulouse, Toulouse, France**
[3] **Université de Toulouse, INP 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 different abstraction levels. 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, tracing, middleware, model, call-tree, graph transformation, CORBA, multithreading

1 Introduction

Mission-critical applications are increasingly assembled from third party components whose quality is only partially controllable [1, 2]. As a result developers must harden their system to guarantee their dependability, usually by adding fault-tolerance mechanisms. This kind of extension, however, requires that they thoroughly understand how their system's internal components behave and interact. Developers must be able to analyse state entangling, capture and restore consistent application states, and track causal dependencies (to allow them to control non-determinism), just to name a few issues. This kind of analysis requires trustworthy and up-to-date information, and as a result, developers often have to reverse-engineer the software they use.

Software reverse engineering has been intensively studied. A major challenge that proposed approaches have had to address is the sheer complexity of the underlying data, in particular when observing a program at run-time. Complex software systems comprise many layers (OS kernel, system libraries, middleware, GUI, *etc.*), each obeying its own logic. Every layer usually interacts with its neighbours in specific and hidden ways. At run-time, their activities tend to overlap, creating 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 emerge [3, 4, 5], the task generally remains cumbersome and error-prone.

A second challenge faced by dynamic reverse engineering (i.e. using run-time data) is the cost of obtaining fined-grained run-time information about a program's execution. Exhaustive tracing, while optimal in terms of coverage, is often far too costly even on medium-size systems.

To address these two challenges, this paper proposes an approach to construct behavioural models of complex multi-layered systems while minimising observation costs. Our prototype, COSMOPEN[1], combines a partial and hence cheap observation technique with a simple but powerful language for graph transformation. Because it minimises intrusion and relies on a flexible analyser, our approach can be applied to a wide range of industry-grade platforms, which we illustrate by reporting on the behavioural analysis of commercial multi-threaded CORBA ORBs.

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

Our approach highlights the possible trade-offs between observation costs and model accuracy in complex software platforms. More precisely we show that even with an almost minimal observation strategy, useful and relevant information can be extracted from running programs. We show that our approach is scalable and can be applied on complex industry-grade platforms.

The remainder of our article is organised as follows: we first discuss the challenges raised by the dynamic reverse engineering of large multi-layer software (section 2). We then introduce two motivating examples that illustrate CosmOpen's core capabilities (section 3). We move on to describe CosmOpen's implementation and transformation language (section 4). Section 5 presents the application of our tool to three popular Object Request Brokers (ORBs): ORBacus, omniORB, and TAO, with a detailed case study of ORBacus. We review related work in section 6, and conclude in section 7.

2 Problem statement

Our interest in reverse engineering tools goes back to our work on fault-tolerance provisioning in complex software platforms [6, 7, 8, 9]. For economic reasons, pre-existing software components (OS, libraries, virtual machine, middleware) are increasingly used in mission-critical applications (railways, avionics, automobile, space exploration, finance, telecommunication). As a result, the developers of such systems face the following two key challenges:

1. Because most components are not specifically developed with fault-tolerance in mind, extra mechanisms are required to harden the resulting systems. To this aim, developers must understand how each component can threaten the overall system, and conversely how the system's overall robustness can emerge from each component's behaviour.
2. Due to the complexity of modern system development, teams focussing on different concerns needs to work independently. Developers need to address fault-tolerance without interfering with the development of the rest of the system.

We have proposed to address those challenges by adapting a well-known architectural paradigm called *computational reflection* to the specificities of large and complex systems [6, 7]. We found that a key step to address the above points is to precisely understand how each component contributes to a system's overall properties. Unfortunately, understanding how a component can influence the dependability of a large system requires a precise analysis of this component, both in terms of structure and behaviour. This is an extremely complex task and led us to look for a reverse engineering approach that would be:

- **Dynamic:** While structural information was needed, our main interest lied in the dynamic behaviour of the system.
- **Non-intrusive:** We did not want to instrument components and libraries, to avoid costly customisation and support the use of COTS (Commercial Off The Shelf).
- **Cheap:** Because of the size of our target systems (over 100,000 LoC per components), we wanted to observe them with a fine granularity, while maintaining an acceptable performance.
- **Flexible:** For performance reasons, the approach should be able to adapt its granularity of observation during the different phases of a program's execution.
- **Discriminative:** The approach should 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 traditional reverse-engineering techniques proposed so far seemed to fit all these requirements: tools for structural analysis that use graph manipulation do not scale well to behavioural data, while tools for behavioural analysis that employ aggregative techniques tend to be unsuitable for program comprehension (figure 1).

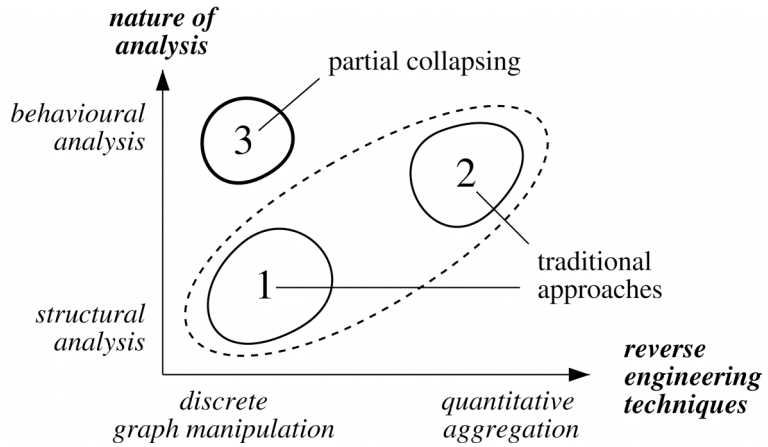


Figure 1: A classification of reverse engineering approaches

Tools in the first category (area (1) in figure 1) rely on graph operations and navigation to help a user encompass the structural complexity of a program. *Ciao* [4,10,11], *GUPRO* [3], and *Rigi* [12,13] 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 (area (2) in figure 1) use aggregative metrics (adding up the times a processor spends in a given function for instance) to condense quantitative data about a program's execution. 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` [14], are prime examples of this approach. Quantitative aggregation of execution data is very useful to identify performance bottlenecks (hot spots) [15], or to profile resource consumption [16], but it removes too much information to convey much insight about a program's internal logic [17].

Graph manipulation has not traditionally been used for behavioural analysis because dynamic events are difficult to represent intuitively in a tractable discrete structure. Only recently pattern-based approaches have been proposed to represent large behavioural data sets as graphs, and thus overcome this complexity lock. The base idea, which we will term *partial collapsing*, is to use patterns to "dose" the collapsing of the behavioural information into a more compact representation (area (3) figure 1). By selectively folding together recurring patterns, 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 [17]. This "pattern-induced" collapsing of subtrees has also been used by Pauw et al. to help locate memory leaks in Java programs. Their technique groups objects according to their class *and* the other objects they refer to [18]. By compacting reference relationships into patterns, they help the user encompass complex object graphs, while maintaining enough information to discriminate objects according to their situation of referencing.

In this paper we propose a new partial collapsing technique to analyse the behaviour of complex multi-layer platforms. Unlike the pattern-based approaches we have just described, we do not use the recurring patterns induced by local programming structures (loops, recursions, *etc.*). Instead, we take advantage of the layered structure of complex software systems. Our prototype tool (CosmOpen) combines two key techniques: i) it uses an inexpensive event extraction scheme, which specifically targets macroscopic interactions within a complex system; ii) it provides a graph manipulation language, which allows users to focus on the level of abstraction they are interested in. In a way, this graph manipulation language acts as a *logical lens* that adapts its focal length to produce a crisp view of a system's dynamics from a blurred set of raw data.

The general philosophy of our tool is similar to that of modern ground telescopes [19]. 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 or by architectural complexity).

3 Introducing COSMOPEN: two motivating examples

We briefly present COSMOPEN's architecture, and then move on to illustrate the key intuitions behind our work with two small examples. The first example is entirely invented and very basic. It illustrates how we lower the cost of dynamic observation while maximising the yield of the resulting data. The second example considers a small multithreaded program in which the collected data contains entangled elements from both the underlying multithreading library and the program itself. We show how COSMOPEN is able to untangle those elements using graph transformation.

3.1 COSMOPEN's overall architecture

COSMOPEN's architecture follows the general guidelines proposed by Chen *et al.* for large source code repositories [4] (figure 2):

1. Raw observation data is extracted from the target program. COSMOPEN can handle structural and behavioural data, although we focus here exclusively on its behavioural capabilities.
2. The raw data is translated into a machine-friendly XML dialect.
3. COSMOPEN's graph transformation engine is used to construct higher-level models from the observed data. The underlying scripting language allows the construction of reusable filters that are well adapted to the layers 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 [20].

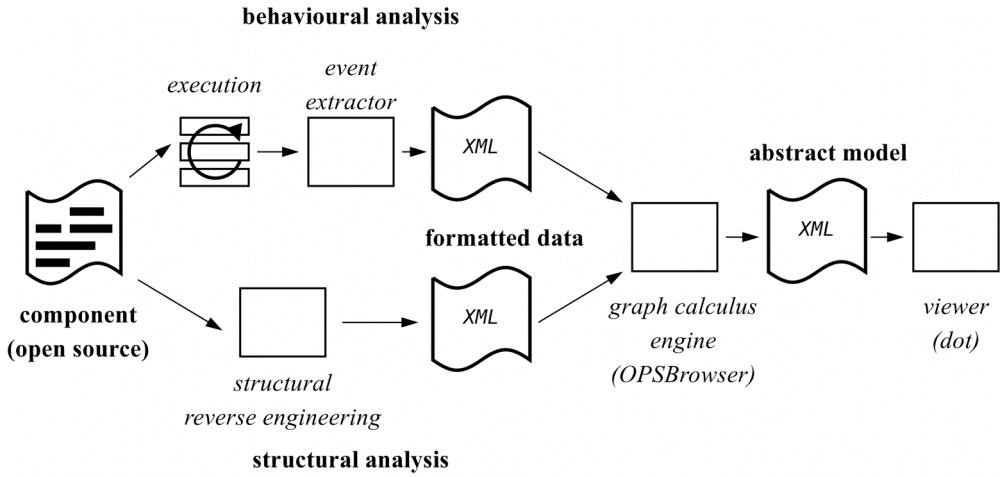


Figure 2: COSMOPEN's general architecture

3.2 Controlling observation costs: a basic example

The problem

Consider the users of an imaginary broadcasting middleware who wish to add fault-tolerance mechanisms to their system. To do so, they need to understand how broadcast requests are proc-

essed. Unfortunately they know nothing about the middleware's structure, except that it provides a `broadcast` primitive and uses the `send` primitive of the underlying OS (figure 3).

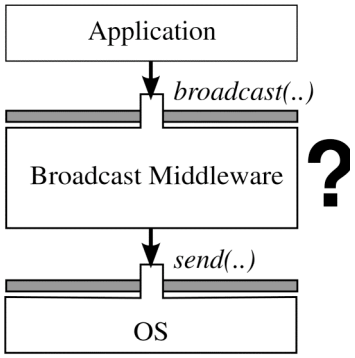


Figure 3: A simple multi-layer reverse-engineering problem

One way to analyse the middleware's behaviour could be to trace all run-time invocations. Many tools such as open-compilers [21], binary code manipulators [22, 23], and aspect-oriented systems can be used to this aim. In COSMOPEN we simply rely on the tracing capabilities provided by modern debuggers such as `gdb` [24]. Unfortunately, exhaustive tracing can be extremely costly. For the CORBA products we consider in section 5, the middleware would not even execute properly when traced exhaustively due to timeout watchdogs.

Addressing the observation bottleneck: foliage and rootage

In COSMOPEN we address this observation bottleneck in two ways:

1) **Stack-trace captures:** Most tracing frameworks, and debuggers in particular, allow 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, and thus captures *multiple* invocations at the cost of *one* interception. We use this insight to lower tracing overheads.

2) **Minimised observation footprint:** Although the developers of our example know nothing of the middleware's internal implementation, the external interfaces of the middleware 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 guide our observation effort. We call this subset *an observation footprint*. The rationale behind this approach lies in the organisation of layered software (figure 4). A software layer (an ORB, an OS kernel) acts as a broker between its surrounding upper and lower layers. It imports lower-level programming primitives and implements 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 well-chosen observation footprint we can follow this "roots-and-branches" structure and reconstruct a view of a layer's internal workings.

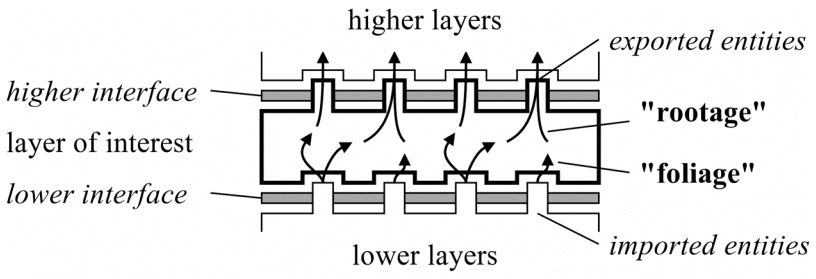


Figure 4: Leafage and Rootage in multi-layer system

Our approach combines the two above insights and only intercepts 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 point below). Let's assume that developers first want 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 uncover internal methods and functions. In our toy example, the first interception 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 an 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 (figure 5). Pending calls are represented as nodes, and nested invocations are shown with directed edges. Calls are labelled with sequence numbers (here from 0 to 4) that reflect the order in which they were recorded. We recognise the documented API `BroadcastEngine::broadcast` (call 2). Call 3 is internal to the middleware and reveals a new method, `BroadcastEngine::marshallAndSend`, which was so far unknown to us.

In most cases, `send` will be invoked several times by the middleware, thus yielding more than one trace. Imagine our middleware makes a very economic use of `send` and we only capture a second trace, identical to the previous one. COSMOPEN is able to merge those two traces in one call-tree (figure 6).

Constructing a call-tree from stack traces raises a number of issues. Most notably, the reconstructed tree is usually not unique. For instance, the tree 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 have solved this by choosing to construct the *smallest call-tree* that is compatible with the observed traces. We return to this issue in section 4, when we discuss COSMOPEN's algorithms for call-tree construction.

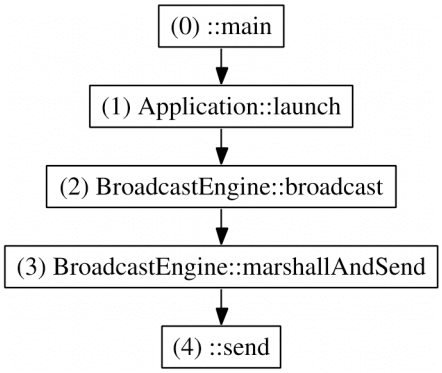


Figure 5: Graphical representation of the first captured trace

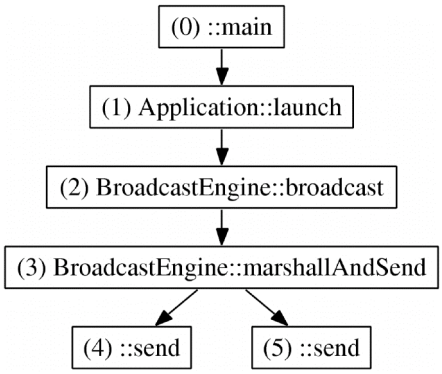


Figure 6: Call-tree resulting from the merging of two traces

Enriching observation footprints

The above call-tree is very basic. It only captures a small facet of the system's behaviour. Developers can easily learn more about the middleware by enriching their observation 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 invocations to `malloc`[2]. Doing so gives us four 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-tree shown in figure 7.

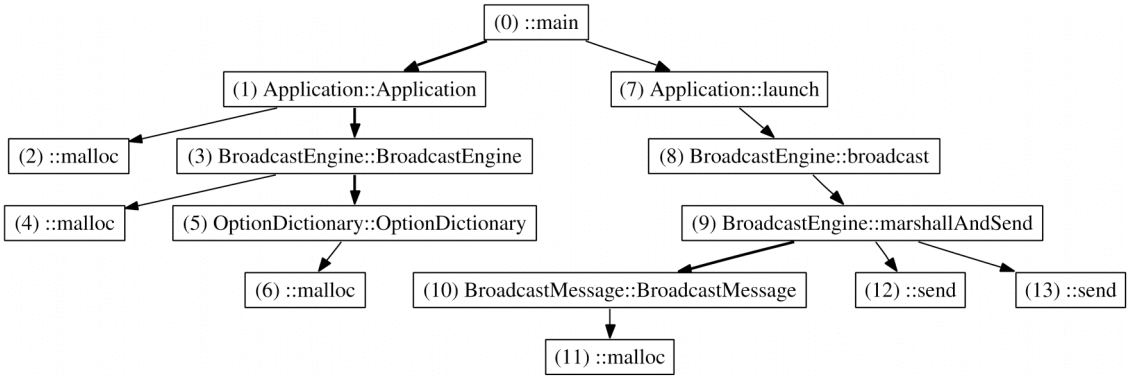


Figure 7: 6 stack traces combined from two OS calls (`send` and `malloc`)

This figure illustrates a limitation of call-trees for the representation of the behaviour of large programs. Our example is small and only traces two OS system calls (`send` and `malloc`). However the resulting call-tree, with 14 nodes, barely fits on the page. Because larger software produces far bigger call-trees (typically with thousands of nodes for the CORBA implementations we consider in section 5), COSMOPEN offers a more compact representation, termed *class interaction diagrams*. This representation is adapted from the object interaction diagrams commonly found in modelling languages such as UML [25]. This is shown in figure 8, with COSMOPEN's

[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 if the middleware executed on top of a virtual machine with its own memory management.

graph manipulation tool running in a console, and the current graph under manipulation (corresponding to the call-tree of figure 7) displayed in a viewer window[3].

In a class interaction diagram, each node represents a class or a standalone function. The sequence of invocations can 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).

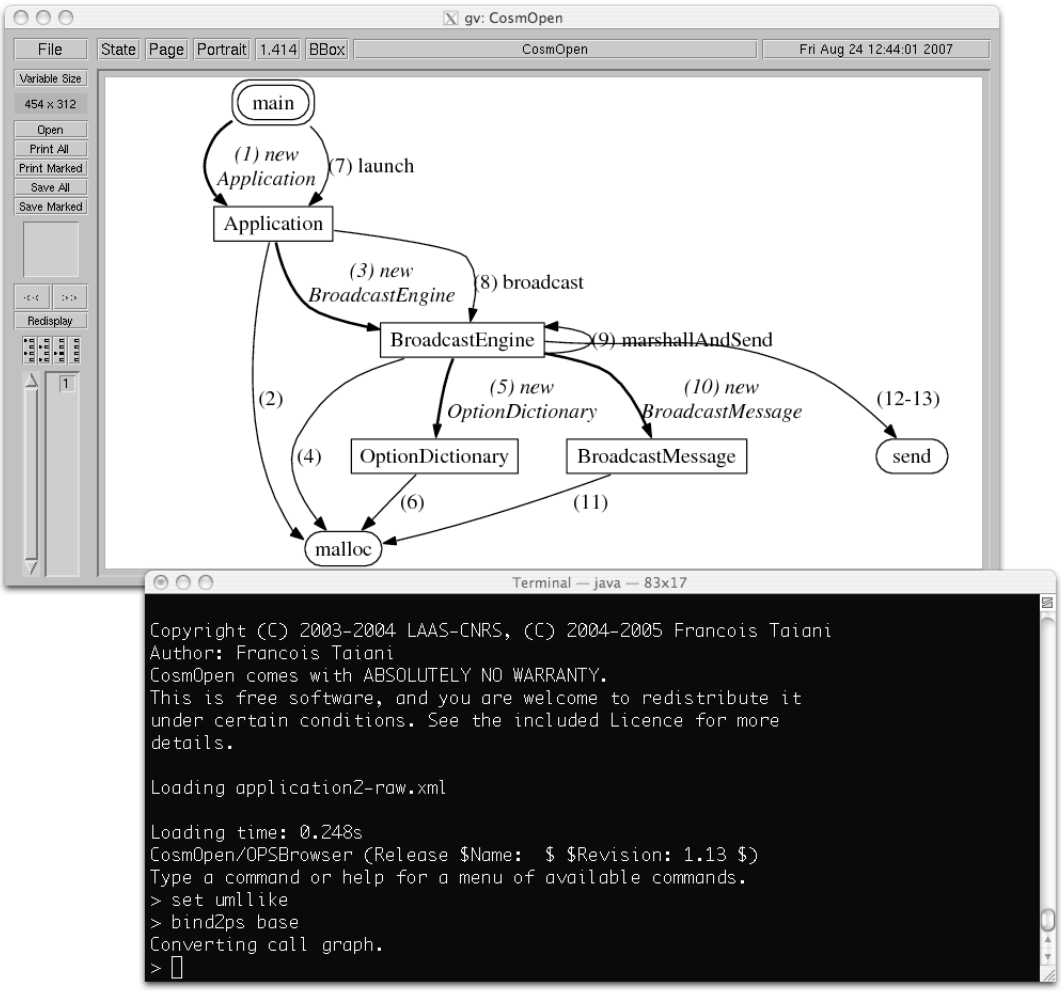


Figure 8: COSMOPEN's graph navigation in action: the call-tree of figure 7 represented as a class interaction diagram

Because sequence numbers can be difficult to follow on large graphs, COSMOPEN offers an alternative labelling scheme that labels edges locally in the order in which they leave and enter a particular node. This is shown on figure 9 for node `bar`: incoming edges are labelled with letters ('a' and 'b') that reflects the order in which `bar` is invoked. Outgoing edges are labelled by a combination of one of the two previous letters and a number (here 'a:1' and 'b:2'). The letter indicates the incoming invocation that triggers the outgoing call (e.g. here the call from `bar` to `foo` is a result of the call from `main` to `bar`); the number indicates the order in which outgoing calls were observed on this particular node. This numbering is only applied to nodes for which an ambiguity

[3] Our prototype uses a simple but effective interactive approach: users can ask the graphs being manipulated to be mirrored in an underlying PostScript file (`bind2ps` in the figure). Using an appropriate viewer (e.g. `ghostview` with the "watch file" option on), users can then visualise their manipulations as they happen.

exits (so here only `bar`). Finally, incoming invocations that have no nested calls are flagged with a filled dot rather than an arrow (such as here the invocation of `tar`).

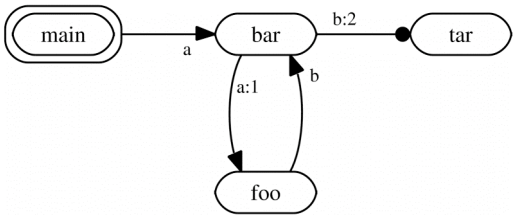


Figure 9: Alternative edge numbering for large graphs

Interaction diagrams are far more compact that standard call-trees, which makes them ideal to represent large call-trees. They also allow developers to relate invocations made on the same class that would otherwise appear completely disconnected, as for instance in figure 8 the creation of the `BroadcastEngine` object — invocation *(3)* — and the invocation of the method `broadcast` on this object — invocation *(8)*.

Unfortunately interaction diagrams alone are not enough to clearly represent oversized trees. As we have just done with `malloc`, developers would typically add more OS calls to their observation footprint. In doing so they are likely to collect information irrelevant to their purpose (i.e. adding fault tolerance to the system), or capture interactions that belong to different logical planes. This data will in turn add unnecessary complexity to any graphical representation, and should ideally be removed. For instance `malloc` belongs to the C++ implementation of our compiler. We might want to remove it from our graph, and only keep information about object construction. We might also want to remove the class `OptionDictionary` since it is not primarily related to the handling of broadcast requests. CosmOpen offers a principled approach to these concerns by providing an interactive graph manipulation language.

3.3 Untangling observed data: a multi-threaded example

To illustrate CosmOpen's graph manipulation abilities, we now turn to the program of figure 10. This program creates two POSIX threads and terminates. The two threads each execute dummy functions (not shown) and exit. Our goal is to observe this program when it runs on Linux (kernel 2.4), and show how CosmOpen can be used to separate OS-level mechanisms from the application logic.

On Linux 2.4, this program executes in a layered environment (figure 11). It uses a user-space library (`pthread`), which in turn uses low-level kernel system calls, such as `clone` (creation of a OS-level "process"[4]), `read` (I/O), or `pipe` (IPC) to provide a POSIX-compliant API.

```
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 10: **An elementary multi-threaded program**

[4] `clone` creates a kernel thread, or light weight process (LWP). In Linux, on most architectures, there is a 1:1 relationship between LWPs and POSIX threads. This is not always the case in other operating systems.

In the following, we show how the program's behaviour can be reconstructed even when tracing solely the two application-level functions `dummy1` and `dummy2` and a set of low-level OS system calls (`read`, `write`, `pipe`, etc.). We will here assume that we know nothing of the `pthread` library. As most of the observation footprint is located *below* the library (see figure 11), our main challenge will be to disentangle the library's execution from that of the program.

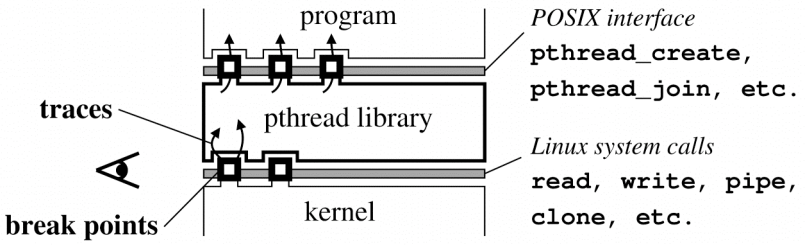


Figure 11: Multithreading in GNU/Linux 2.4

This example is of course oversimplified since we can tell what the program does by simply looking at its code. POSIX is also a well-known standard and we could trace its API directly. The example is however representative of large and complex software, where a manual inspection of the code rapidly becomes unpractical, and intermediary libraries are often little documented, if at all.

3.3.1 Obtaining a first interaction diagram

A dynamic observation of the program using the above breakpoints yields 14 stack traces, from which COSMOPEN constructs a 28-node call-tree, shown in figure 12 (call-tree) and in figure 13 (interaction diagram)[5]. At the top of figure 13 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.

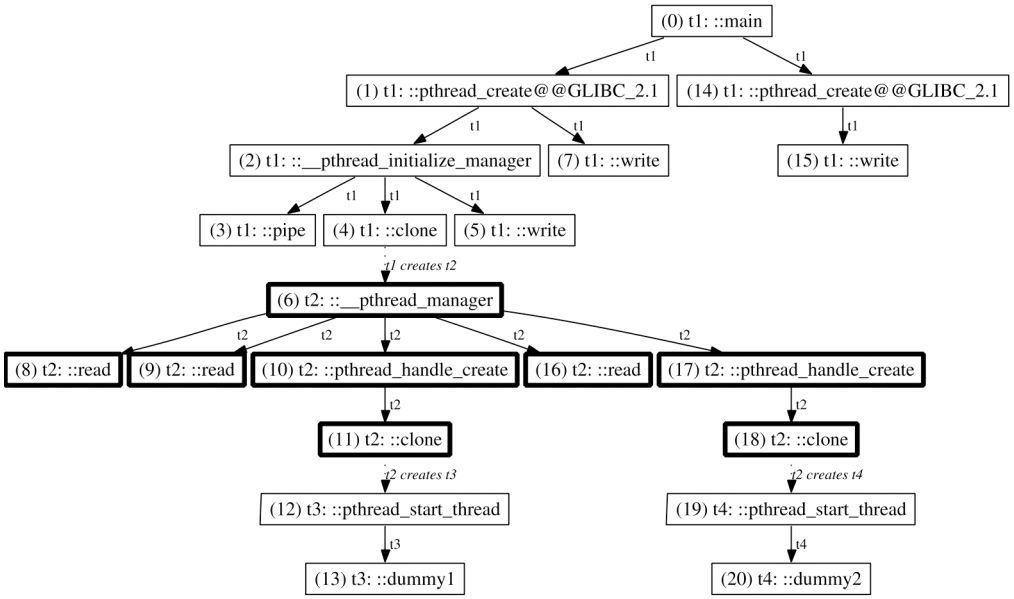


Figure 12: Thread Creation in `libpthread.so` in GNU/Linux 2.4 (raw call-tree)

[5] In figure 13 the layout of the node with `dot` has been optimised to highlight the different layers involved.

The captured traces belong to multiple threads, some of which are dynamically instantiated. From `gdb`'s output, COSMOPEN can reconstruct how each thread was launched. (We discuss in section 4 how this is done.) 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. On both graphs, we have highlighted the activity of thread t2, whose role we discuss just below.

These two diagrams raise a number of observations:

- First, although the original code is very small, the reconstructed call-tree 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 observe four (t1 to t4).
- Finally, the diagram does not reflect the structure of the reverse engineered program: the fact that the main thread t1 creates two threads is not apparent.

The fundamental reason for these three points is that we have captured 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*: the diagram generated by COSMOPEN contains two logical planes, yielding a confused picture of the program's actual behaviour.

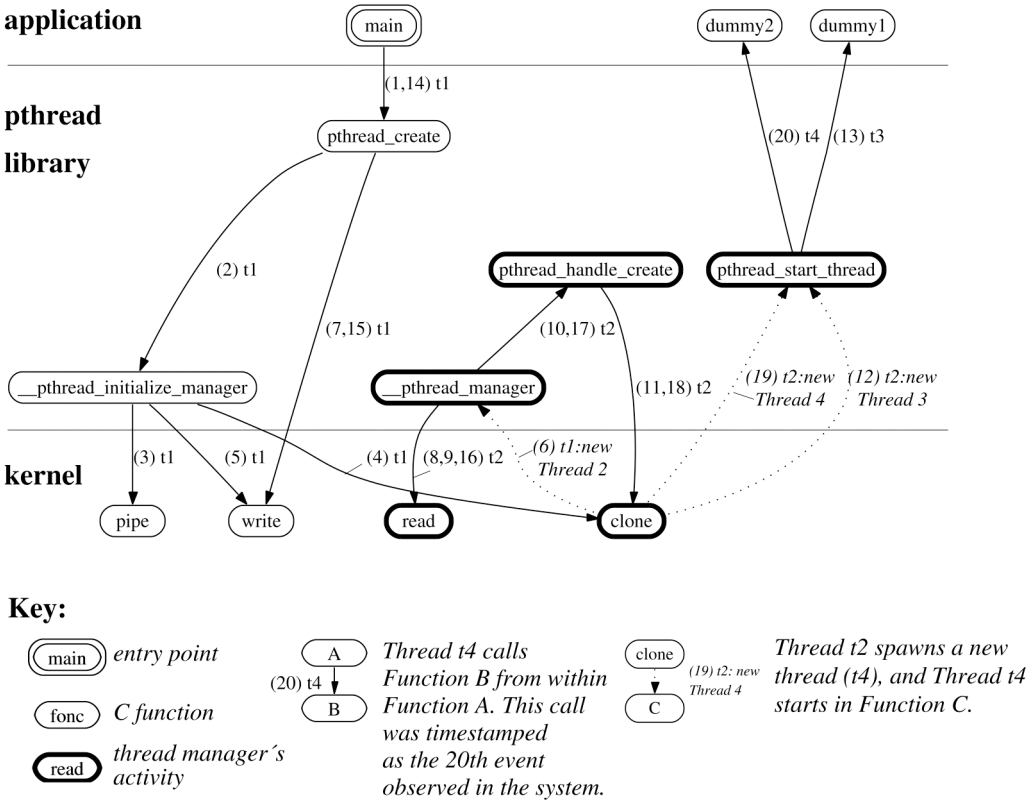


Figure 13: **Thread Creation in `libpthread.so` in GNU/Linux 2.4 (interaction diagram)**

3.3.2 Multithreading under Linux: the raw call-tree explained

Before we discuss how COSMOPEN can be applied to this example, we must first explain what is happening in figure 13. As in figure 8, figure 13 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. `pthread_create` is called twice by t1, which is represented by two sequence numbers: (1,14), meaning that the second invocation has the number 14. From `pthread_create`, t1 invokes `__pthread_initialize_manager` (tagged 2). t1 then proceeds to invoke three system calls: `pipe`, `clone`, and `write` (tagged 3, 4, and 5).

With these last three calls t1 is initialising a special thread called the *thread manager*[6] (t2 on the diagram) that is internal to the pthread library: 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. The launch of t2 is represented by a dotted arrow from `clone` to `__pthread_manager`, and tagged as the 6th observed event.

After executing invocation 5 to `write`, t1 returns from `__pthread_initialize_manager` and 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 (invocations 8 and 9 to `read`). t2 then creates a POSIX thread t3 that goes on to execute `dummy1` (invocations 10, 11, 12, and 13). The creation of a second thread by t1 follows the same path from invocation 14 onward.

Admittedly, the above insights into the innards of the multithreading library cannot be inferred solely from the interaction diagram. Rather the diagram constructed by CosmOpen offers a high-level representation of the key interactions between the library's parts. We used this information to guide our exploration through the sources of the library (for instance by looking at the implementation of `__pthread_initialize_manager` to understand its pipe mechanism), and to search information on the web (in mailing list archives essentially). During these activities, CosmOpen's provided a bird-view map of the library's behaviour, telling us what mechanism to analyse and which part of the code to investigate. It thus became the key driver of our analysis.

3.3.3 Separating logical planes with CosmOpen

The call diagram shown on figure 13 contains two overlapping logics: the low level workings of the `pthread` library, and the higher-level logic of the 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 sharpen this representation by filtering out the multithreading library. In the earth-bound telescope metaphor of section 2, the ability to discriminate between different abstraction planes is our way to compensate for cheap observation techniques and for a lack of knowledge on the internals of a system's components.

Abstracting the multithreading library really means abstracting thread t2. t2 acts as a proxy to perform POSIX threading operations. Interaction between t2 and other threads (in our example t1) occurs through a local RPC mechanism based on a pipe IPC primitive. For instance when t1 creates t3 inside `dummy1`, it sends a request to t2 (invocation 7 to `write` in figure 13). The actual creation of t3, from the point of view of the OS, happens when t2 invokes `clone` (invocation 11 in figure 13) after it has read t1's request from the pipe (invocation 9 to `read`).

To abstract away t2, we must transform the previous sequence of *observed* operations into some higher-level *abstract* event (t1 creating t3). We further want this transformation to be *automatic*, to apply it to potentially very large call-trees.

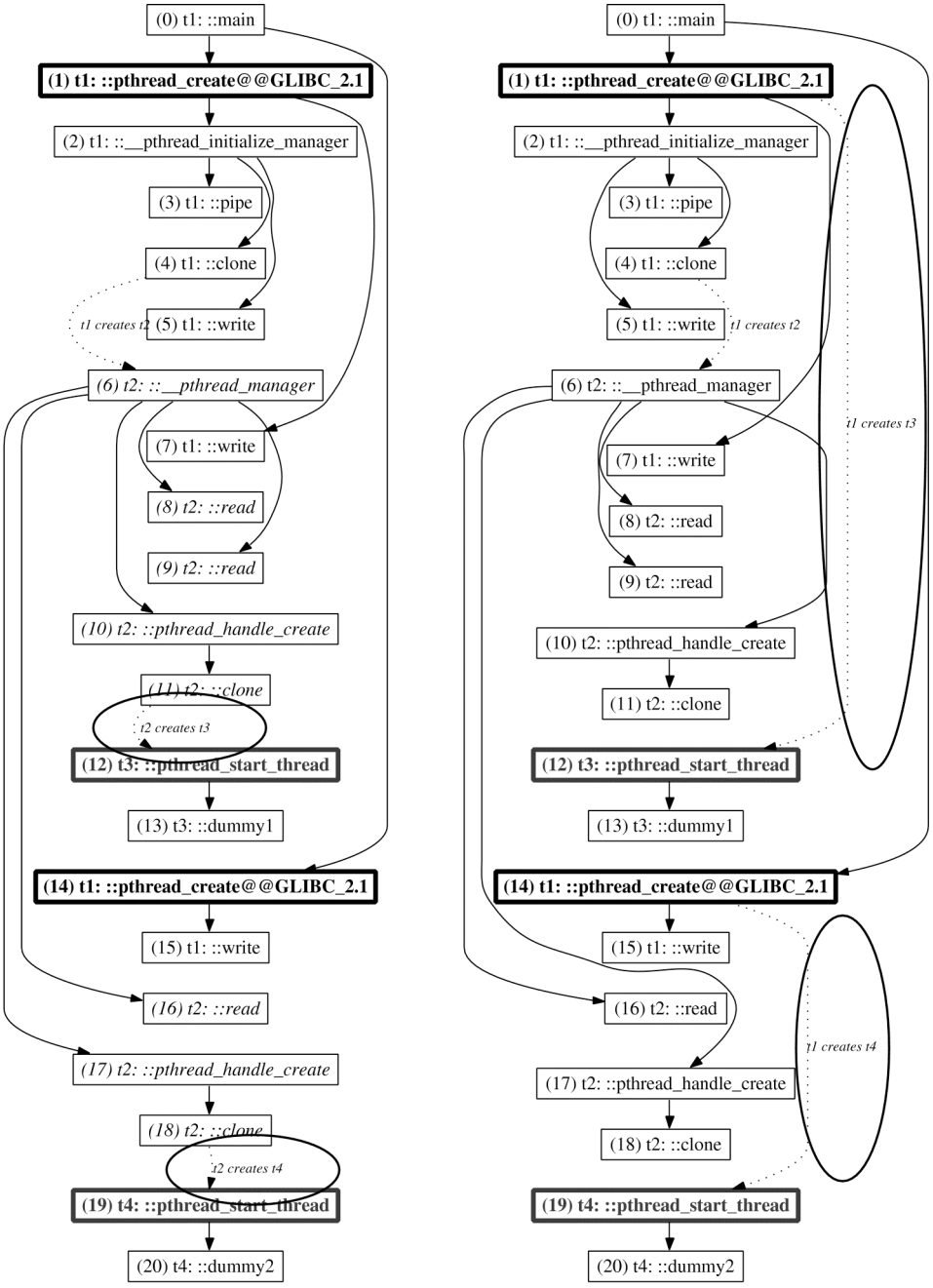
Our goal, in the remainder of this section, is to show that this transformation can be achieved by a sequence of simple graph operations, and that these operations are *generic*: they can be applied to any program using the `pthread` library, independently of the program's higher-level logic.

3.4 Lost data dependencies

In our example the program semantics we wish to represent—the main function creating two threads—is not captured in the call-tree 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 threads t3 and t4. The causal link does exist: t3 and t4 are created because t1 writes on the pipe shared with t2. However, since we only trace invocations, we lose this data dependency. Because

[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 [26] for details.

interaction diagrams only represent a program's control-flow, they cannot capture the data flows that occur through shared structures. We need to reconstruct this causal relationship from the clues left in the call-tree we have at hand.



(a) Our program's call graph as a timeline **(b)** The timeline after the `fuse` operation

Figure 14: The `fuse` operation applied to our program's call-tree

Fortunately, we can translate our understanding of the `pthread` library in terms of a simple graph transformation. The relationship between `pthread_start_thread` and `pthread_create` obey a rather obvious property that we have termed *pair-wise sequencing*. 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`: the first `pthread_create` with the first `pthread_start_thread`, etc.

As CosmOpen tracks the order of invocations, we can identify for each new thread the invocation to `pthread_create` that created it. Figure 14-a illustrates this on our example. It shows exactly

the same call-tree as figure 13, except that nodes (e.g. invocations) have been ordered on a timeline according to their sequence numbers (from top to bottom). Four invocations are shown in bold: the two invocations to `pthread_create` by t1 (invocation 1 and 14) and the two corresponding invocations to `pthread_start_thread` (12 by t3, and 19 by t4).

Using pair-wise sequencing, we immediately see that invocation 1 must be associated with invocation 12, and invocation 14 with invocation 19. The call-tree can be manipulated to reflect this new understanding by changing the parent node of both invocations 12 and 19, as shown on figure 14-b: two new edges now connect invocation 1 with 12, and 14 with 19. This transformation is supported by COSMOPEN through a generic operation called *fuse*. *fuse* takes two sequences of nodes (a_1, ..,a_n) and (b_1, ..,b_n) — here (1, 14) and (12, 19) — and removes each b_i and its subtrees from its current position in the graph and makes it a child of the corresponding a_i. (See the appendix for a formal definition of *fuse*.) *fuse* can be used whenever a particular interaction is hidden in a temporal sequence of events. This applies for instance to signal handlers, provided the raising of a signal and the execution of a handler are being traced through breakpoints.

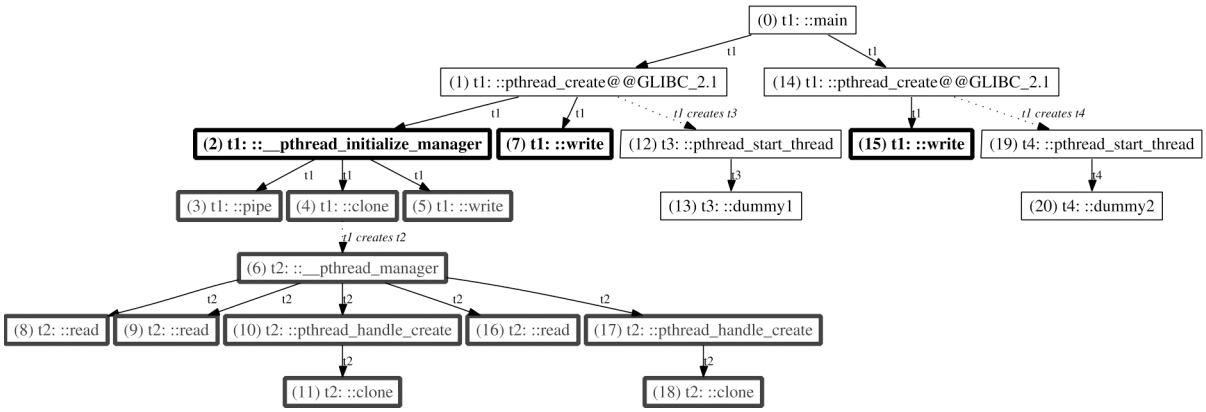


Figure 15: The timeline of figure 14-b represented as a traditional call-tree

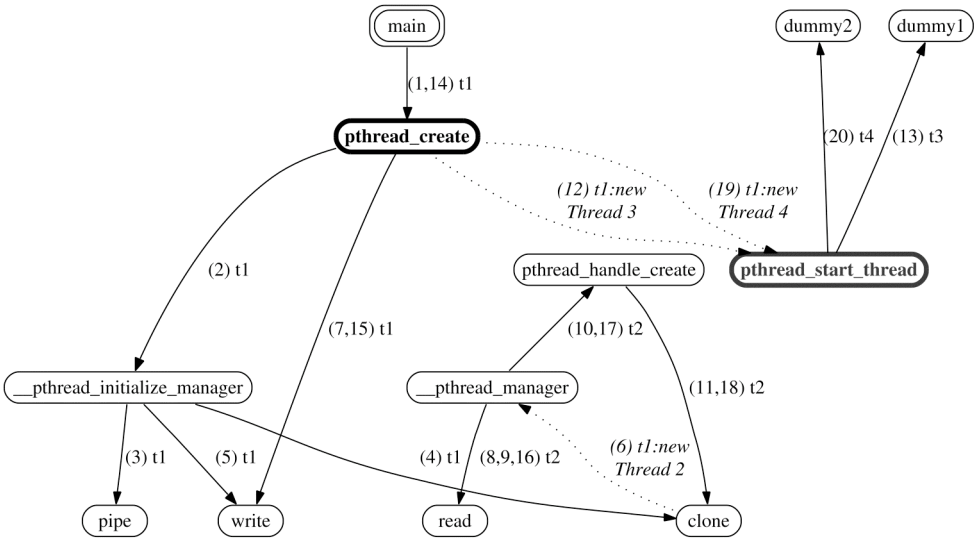


Figure 16: The timeline of figure 14-b represented as an interaction diagram

There is, however, a difficulty that we have so far silently swept under the rug: in pathological runs, some thread creations might fail. When this happens, COSMOPEN will register an invocation to `pthread_create`, but no corresponding thread will start, and our pair-wise matching could create erroneous associations. COSMOPEN prevents this by aborting the *fuse* operation when it detects there is not the same number of "parent" and "child" nodes. Similarly *fuse* gets aborted if

the two sequences of nodes fail to alternate on the time line (i.e. a_1, b_1, a_2, b_2, etc.). By guarding certain graph transformations, we are ensuring that the produced results are always consistent. This conservative approach is linked to low-cost observation: cases may arise in which the limited clues we have gathered are insufficient to accurately analyse a program. This seems a reasonable price to pay, and it actually never happened in the case study we present in section 5.

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

3.5 Eliminating the threading library

We have progressed in our attempt to reconstruct the original program's logic but we still need to eliminate from the call-tree all invocations that are internal to the threading library (represented with a thicker frame on figure 15). We could do this by manually removing individual nodes. Unfortunately this transformation would not be generic. Instead, we need a general way to describe what must be removed even if thread t1 created 10 threads, or if 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, the closure can be removed from the whole graph, resulting in figure 17.

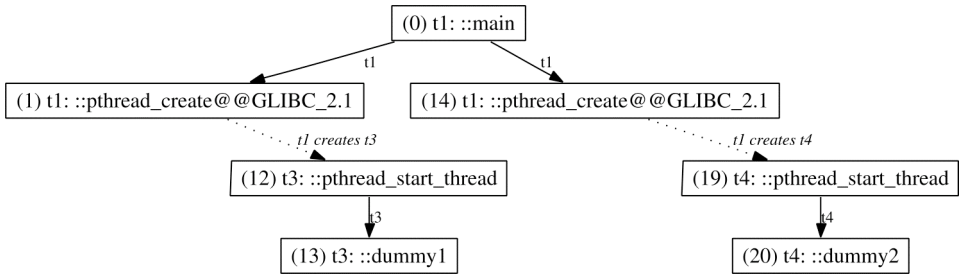


Figure 17: The call-tree of figure 16 once internal pthread calls have been removed

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 (`dummy1` and `dummy2`) attached to the call-tree. The resulting diagram (figure 18) 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.

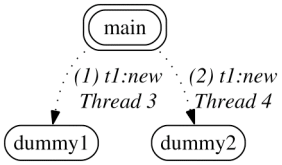


Figure 18: Final interaction graph, after abstracting the remaining POSIX functions

3.6 Conclusion

The actual COSMOPEN commands required to perform the sequence of operations we have just described is shown in figure 19 below. We discuss the meaning of these commands in more detail

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 '*' is a wildcard 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 is not specifically linked to the small program we started with, but instead can be re-used as a 'lens' 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 easily reused across programs.

4 COSMOPEN: reconstructing call-graphs from stack traces

COSMOPEN allows developers to reverse-engineer complex software at an affordable cost thanks to two main components: (i) its *dynamic event extractor*, and (ii) its *graph manipulation engine*. The event extractor relies on a cheap observation approach to reconstruct a call-tree from stack traces, while the graph manipulation engine provides a simple yet powerful scripting language to analyse the resulting data.

4.1 Observation on a budget: from stack traces to call-trees

COSMOPEN obtains behavioural information of a program by gathering stack traces on key interface points (the "observation footprint"). This approach works on any platform where calls can be intercepted dynamically and the content of the current thread's stack captured as output[7]. These are standard features available in almost all debuggers, and increasingly found in dedicated interfaces, such as the native JVM Tool Interface (JVMTI) [27] 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. Additionally, developers can develop their own customised extractors, as shown in the callout of 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).

[7] COSMOPEN remains blind to any invocation that escapes the debugger. This applies to in-lined functions when these are not supported by the debugging process. (In gdb, support for in-lining varies across platforms.)

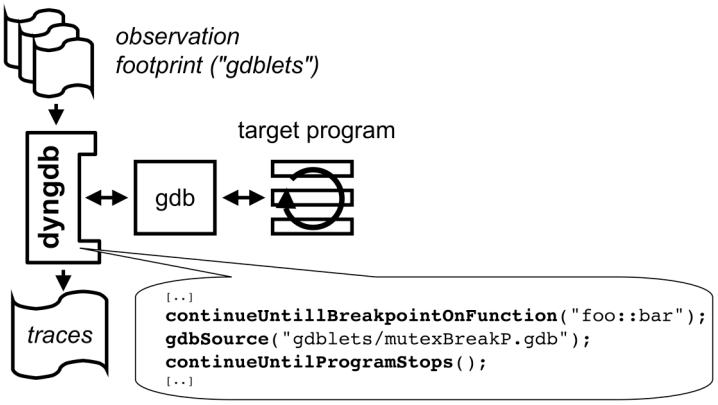


Figure 20: COSMOPEN's Event Extractor for C/C++ (`dyngdb`)

Call-Tree Reconstruction

COSMOPEN's graph engine uses the information contained in the captured stack traces to reconstruct a call-tree that represents the program's behaviour. Unfortunately, as alluded to in section 3.2, several call-trees can be reconstructed from the same stack traces. For instance: in our hypothetical broadcast middleware (section 3.2), the two traces we collected can be converted in up to five call-trees (figure 21[8]): in this particular case the two traces are identical, and the key question is to decide how often each of the involved methods has really been invoked.

The first tree (1) can readily be dismissed: since `gdb` registered two traces, we know that `send` must have been invoked twice. We cannot however tell which of the remaining call-trees is the correct one: `send` might have been called twice, and all other methods called only once (tree 2); or `main` might have called the `launch` method twice, thus triggering twice the same chain of invocation (tree 5), with any tree in-between a valid possibility.

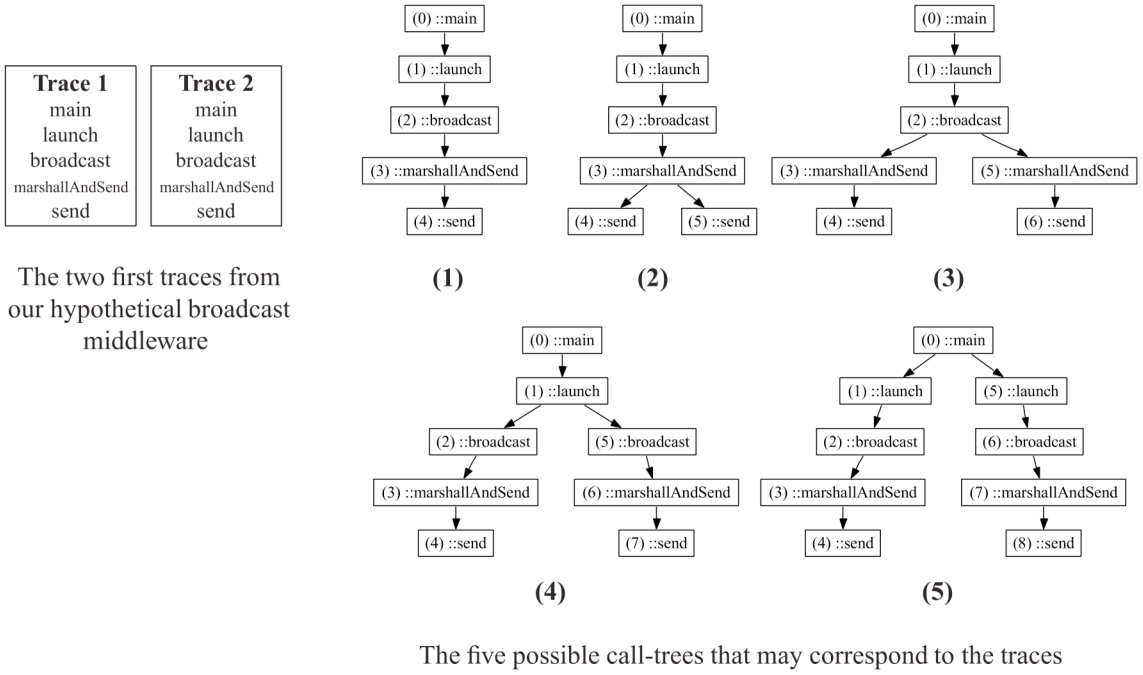


Figure 21: Even the two simple traces of section 3.2 can produce five different call-trees

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

This ambiguity arises because we rely on indirect information—in the form of the invocations of the call-stack—to infer the behaviour of the rest of the program. This approach reduces observation costs, but makes it harder to determine when and how frequently functions were invoked. For instance, in the above example, was `broadcast` invoked once (call-trees 1, 2 and 3) or twice (call-trees 4 and 5)? Because `launch` was not instrumented with an explicit breakpoint, there is no way to tell.

To resolve this ambiguity, CosmOpen always opts for *the call-tree with the smallest number of nodes that is 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 choice reflects two main concerns: i) to capture in the tree as much information as can be inferred from the traces; ii) to only present correct information, in particular regarding the ordering of invocations. Choosing the smallest tree means that nodes only appear to the call-tree when an invocation is guaranteed to have occurred (as for the two `send` nodes above), and that ambiguous cases are resolved using the simplest compatible pattern of invocations (call-tree 2 rather than 3, 4 or 5).

This approach to call-tree reconstruction is embedded in algorithm 1 below. This algorithm reconstructs a call-tree from a set of stack traces collected on a mono-threaded program. The algorithm works on lists and sets. The traces captured in an observation run are represented as a list of stack traces (traceSequence at line 2); and stack traces are represented as lists of symbols. Symbols are the names of the functions and methods appearing in the program ('`main`', '`launch`', '`broadcast`' in our previous example). The call-tree that is progressively constructed is represented a directed graph (V, E), where V contains the nodes of the call-tree, and $\mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$ its edges. Each node is associated with a symbol. **new** $Node(s)$ returns a new node associated with the symbol s, and $symbol(n)$ returns the symbol associated with node n. For example: in call-tree 2 in figure 21, the symbol of node (0) is '`main`', while nodes (4) and (5) are associated with the same symbol '`send`'. New nodes are time-stamped with a running counter that reflects the order in which they are inserted into the tree.

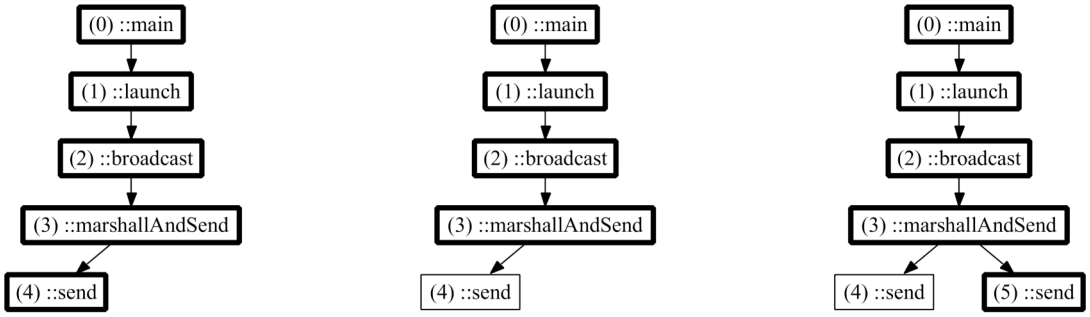
```
 1: ActiveTreePath ← (); V ← E ← ∅
 2: traceSequence = (trace_1, trace_2, .., trace_n)
 3: for i = 1 to |traceSequence| do
 4:     AddTraceToGraph(traceSequence[i], V, E, ActiveTreePath)
 5: end for

 6: procedure AddTraceToGraph(trace, V, E, ActiveTreePath)
 7:     for j = 1 to min( |ActiveTreePath|, |trace| − 1 ) do
 8:         if symbol(ActiveTreePath[j]) ≠ trace[j] then break
 9:     end for
10:     truncate(ActiveTreePath, j)
11:     previousNode ← last(ActiveTreePath)
12:     for k = j to |trace| do
13:         newNode ← new Node(trace[k])
14:         ActiveTreePath ← ActiveTreePath + (newNode)
15:         V ← V ∪ {newNode}
16:         if previousNode ≠ NIL then E ← E ∪ { (previousNode, newNode) }
17:         previousNode ← newNode
18:     end for
19: end procedure
```

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

In algorithm 1, lists are noted (x_1, x_2, …, x_n); () is the empty list; | t | is the length of list t; and t[j] is the jth element of list t. When t is a stack trace, we refer to the pair ⟨ t, j ⟩ as the jth *stack frame* of the stack t and to t[j] as the content of the frame ⟨ t, j ⟩, in reference to the term used in compilers and debuggers. The helper function *last*(t) returns the last element of a list, or the special value NIL if the list is empty. *truncate*(t, k) removes from the list t the elements with an index equal to or greater than k (i.e. *truncate*(t, k) = (t[1], t[2], .., t[k -1])). If k > | t |, *truncate*(t, k) has no effect. Sequence numbers are allocated by the constructor **new** *Node*(..) (line 13) in the order in which nodes are created. The same sequence number is never allocated more than once.

The algorithm works incrementally by processing stack traces in the order in which they were observed (lines 3-5, and procedure *AddTraceToGraph*). When processing a new trace (line 6), the algorithm computes the overlap between the new trace and the ActiveTreePath variable (lines 7-9) to determine which part of the trace corresponds to new invocations. The variable ActiveTreePath contains a list of call-tree nodes that have not returned yet, i.e. that represent pending invocations. The second part of *AddTraceToGraph* (lines 11-18) updates the call-tree by inserting the tail of the trace that did not overlap with ActiveTreePath into the graph (V, E), and updates ActiveTreePath along the way.

By preventing symbols already present in ActiveTreePath to be added as nodes to the tree, the algorithm ensures the constructed tree is minimal (we return to this point below). At the same time, because this overlap does not consider the last symbol of a trace (| trace-1 | at line 7), the algorithm guarantees each individual breakpoint will be added as an individual node.



(a) after adding trace 1

(b) at line 11, after computing the overlap with trace 2, and truncating **ActiveTreePath**

(c) after adding trace 2

Figure 22: Applying algorithm 1 to the traces of figure 21. The bold nodes represent the content of ActiveTreePath.

Figure 22 illustrates a run of algorithm 1 on the two traces of figure 21. When first processing trace 1 = (‘`main`’, ‘`launch`’, ‘`broadcast`’, ‘`marshallAndSend`’, ‘`send`’), ActiveTreePath is empty. The `for` loop at lines 7-9 does not execute (the overlap is empty), and trace 1 is entirely inserted into the call-tree (lines 11-18, figure 22-a). Because trace 2 is identical to trace 1, the algorithm finds trace 2 to completely overlap with ActiveTreePath (lines 7-9), and only truncates the last node of ActiveTreePath (node (4) at line 10, figure 22-b). The last symbol of trace 2, ‘`send`’, is then inserted as a new node (5) as a child of node (3) (lines 11-18, figure 22-c).

Formal characterisation of the constructed tree

Formally we say that a call-tree (V, E) is *trace compatible* (or *compatible* for short) with a sequence of observed stack traces (trace_1, trace_2, ... trace_n) if and only if there exists a surjective

mapping Φ from the frames $\langle\, \mathsf{trace_i}\,, j\,\rangle$ of the observed traces onto the nodes of V that fulfils the following properties P0, P1, P2 and P3[9].

Property P0: Symbol Consistency

Stack frames are mapped to nodes with the same symbol:

$$\forall\, \mathsf{trace}_i \in \mathsf{traceSequence}, 1 \leq k \leq |\mathsf{trace}_i| : symbol(\Phi(\langle \mathsf{trace}_i, k\rangle)) = \mathsf{trace}_i[k]$$

Property P1: Trace Inclusion

Each observed trace is mapped to a path in the tree that starts from the tree's root:

$$\forall\, \mathsf{t}_i \in \mathsf{traceSequence} :$$
$$\Phi(\langle \mathsf{t}_i, 1\rangle) \text{ is the root of } (\mathsf{V}, \mathsf{E}) \wedge$$
$$\forall\, 1 \leq j < |\mathsf{t}_i| : \Big(\, \Phi(\langle \mathsf{t}_i, j\rangle),\, \Phi(\langle \mathsf{t}_i, j+1\rangle)\, \Big) \in \mathsf{E}$$

Property P2: Breakpoint Discrimination

The last frame of every stack trace (e.g. '`send`' in the previous example) is mapped to a node distinct from those of the stack traces that precede it:

$$\forall\, \mathsf{t}_i, \mathsf{t}_j \in \mathsf{traceSequence} : i < j \Rightarrow \forall\, 1 \leq k \leq |\mathsf{t}_i| : \Phi(\langle \mathsf{t}_j, |\mathsf{t}_j|\rangle) \neq \Phi(\langle \mathsf{t}_i, k\rangle)$$

Property P3: Order Conservation

The sequence numbers of the nodes of the tree reflect the order in which traces are observed. More precisely: if a trace trace_j is observed after another trace trace_i, then the tail of trace_j that does not overlap with trace_i gets mapped to nodes with higher sequence numbers than the nodes of trace_i.

$$\forall\, \mathsf{trace}_i, \mathsf{trace}_j \in \mathsf{traceSequence}, \forall k \in \Big[1, min(|\mathsf{trace}_i|, |\mathsf{trace}_j|)\Big] :$$
$$(j > i) \wedge \Phi(\langle \mathsf{trace}_i, k\rangle) \neq \Phi(\langle \mathsf{trace}_j, k\rangle) \Rightarrow$$
$$\forall l \in \Big[k, |\mathsf{trace}_j|\Big], \forall m \in \Big[1, |\mathsf{trace}_i|\Big] :$$
$$timeStamp\Big(\Phi(\langle \mathsf{trace}_j, l\rangle)\Big) > timeStamp\Big(\Phi(\langle \mathsf{trace}_i, m\rangle)\Big)$$

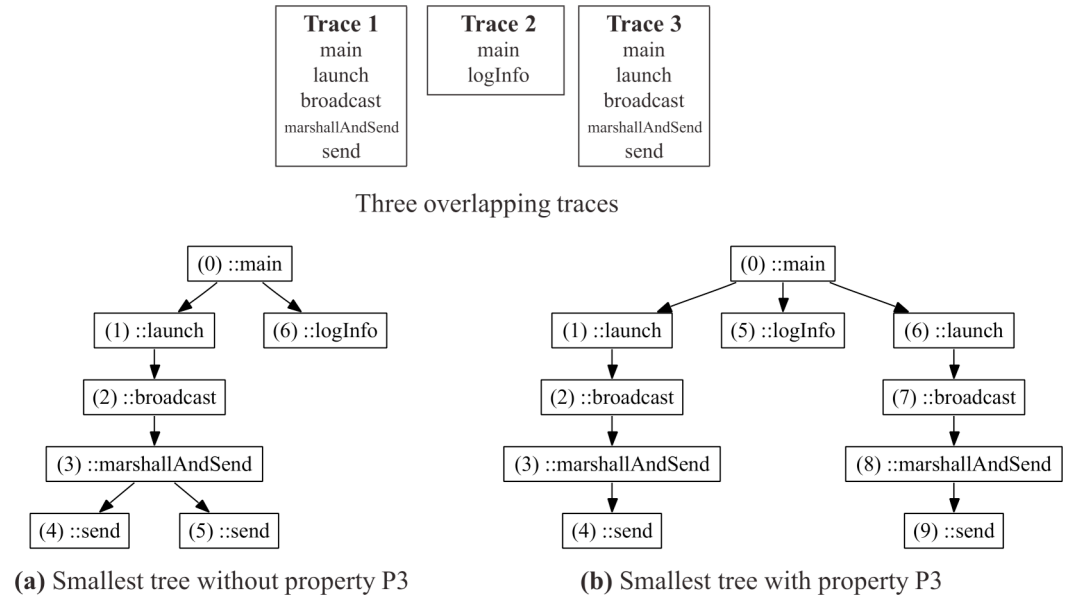
Intuitively, **P0** insures the mapping Φ makes sense, i.e. that tree nodes are labelled with the same symbol that the stack frames that get mapped onto them; **P1** translates the sequence of nested calls within a stack trace into a path of the call-tree; **P2** insures that we discard tree 1 in figure 21 in favour of tree 2 (since in tree 1 the last frame of trace 2, `send`, gets mapped to the same node as the last frame of trace 1, thus violating the property); and **P3** maps the ordering between stack traces onto the nodes of the call-tree. If we return to figure 22, trace 1 gets mapped to the path (0) → (1) → (2) → (3) → (4), and trace 2 to the path (0) → (1) → (2) → (3) → (5), thus fulfilling **P1**. The last frame of trace 1 gets mapped to node 4, and the last frame of trace 2 to node 5 (**P2**). Finally, node 5 has a higher sequence number that node 4 thus reflecting the fact that trace 2 was captured after trace 1 (**P3**).

Although property P3, *Order Conservation***,** might seem superfluous, it is crucial to insure that the correct call-tree is built. For instance in figure 23, an additional stack trace containing an invocation to the function `logInfo` has been observed between the two traces of figure 21. Without property P3, tree (a) would be the smallest tree respecting properties P0, P1, and P2: all traces are contained in the tree, and breakpoint activations are mapped to distinct nodes. Unfortunately, tree (a) implies that all `launch` invocations (be it one or more) happen before `logInfo`. It also does not show that `launch` is invoked twice, although this is clear from the traces. Tree (a) is therefore both incorrect and incomplete. This happens because property P3 is here violated: the second `launch` stack frame of stack trace 3 is mapped to node (1), and is thus ordered *before*

[9] 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 sequence numbers higher than their parents.

node (6) `logInf` of trace 2, in contradiction to the fact that stack trace 3 was recorded *after* stack trace 2.

By contrast, tree (b)—the smallest tree respecting properties P0, P1, P2, and P3—does conserve the ordering of frames. The two `launch` frames (from traces 1 and 3) get mapped to different nodes: one before `logInfo`, and one after it. Property P3 is thus respected. The tree correctly reflects the ordering of invocations, and shows that `launch` was invoked twice.



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

Figure 23: Without Order Conservation property P3, tree (a) would be a valid reconstructed call-tree for the three traces shown above

Using this definition of compatibility, algorithm 1 can be shown to fulfil the following theorem:

Theorem 1:

> The set of call-trees that are compatible with a sequence of observed stack traces (according to the above definition) is non-empty. The call-tree of this set with the smallest number of nodes is unique and is the tree constructed by algorithm 1.

A detailed outline of the proof of Theorem 1 is available as a technical report [45]. For place reasons, we do not repeat this proof here, and we refer interested readers to this report for a more thorough discussion. The proof contains three parts: (i) first, the call-tree $\mathbf{G}_0$ constructed by algorithm 1 can easily be shown to fulfil P0-3 (using line 13 to construct the mapping Φ); (ii) second, any call-tree $\mathbf{G}$' respecting P0-3 can be shown to have at least as many nodes as $\mathbf{G}_0$ (proving the minimality); (iii) and third, any call-tree $\mathbf{G}$' respecting P0-3 that contains the same number of nodes as $\mathbf{G}_0$ can be shown to be equal to it (proving uniqueness). The key to the second and third parts is to construct a surjective homomorphism[10] from $\mathbf{G}$' onto $\mathbf{G}_0$ using the two mappings Φ' and Φ_0 that are implied by P0-3. The surjection shows $\mathbf{G}$' has at least as many nodes as $\mathbf{G}_0$ (i.e. $|\mathbf{G}'| \geq |\mathbf{G}_0|$). It also shows that if $\mathbf{G}$' has the same number of nodes as $\mathbf{G}_0$ ($|\mathbf{G}'| = |\mathbf{G}_0|$) then the two graphs are isomorphic (since a surjection between two finite sets with the same number of elements must be a bijection). This thus shows the *minimality* of the constructed call-tree $\mathbf{G}_0$: any call-tree that verifies P0-3 and has more nodes than $\mathbf{G}_0$ can be 'collapsed' onto $\mathbf{G}_0$ (by way of the surjective homomorphism). It also shows its *uniqueness*: if we assume there exists another tree

[10] We use graph homomorphisms between labelled graphs, with an additional order conservation property on sequence-numbers. See [45] for more details.

that verifies P0-P3 with exactly the same number of nodes, then this graph is equal to $\mathbf{G}_0$ (modulo an isomorphism).

Multithreading

Multithreaded programs are tackled using an extended version of algorithm 1, shown in algorithm 2. This extended version uses information related to the thread-creation API (`clone` on Linux), which must be included in the observation footprint for the algorithm to work.

The algorithm processes a sequence of observations (obs_1, obs_2, … obs_n) which contains both stack traces (as in algorithm 1) and thread creation notifications (as reported by `gdb`). Each observation (be it a trace or a creation notification) is associated with a thread ID. Figure 24 provides an example of such a sequence (in the XML format used by COSMOPEN) for a simple multithreaded program. The algorithm loops through the observations and follows a two-mode approach:

1) If the observation is a trace, the trace is added to the call-tree using the same procedure *AddTraceToGraph*() as algorithm 1 (line 8, the procedure is not repeated here). The algorithm manages a distinct active path for each individual thread (in the array ActiveTreePathsByThread), and thus maintains an individual subtree for each thread of the program.

2) If the observation is a thread creation (such as '`<new thread="2"/>`' in figure 24-b), the algorithm associates the thread just created with the latest '`clone`'[11] breakpoint that it encountered (line 20).

[11] The particular thread creation API (here '`clone`' on Linux) is platform dependent, and is a parameter given to COSMOPEN.

```
1: obsSequence = (obs_1, obs_2, .., obs_n)
2: ActiveTreePathsByThread ← ((), (), ..); V ← E ← ∅
3: knownThreads ← {1}                                          ▷ main thread
4: threadSpawningPoints ← (NIL, NIL, .. )
5: latestClone ← NIL
6: for i = 1 to |obsSequence| do
7:     k ← getThreadID(obsSequence[i])
8:     if isTrace(obsSequence[i]) then
9:         AddTraceToGraph(obsSequence[i], V, E, ActiveTreePathsByThread[k])
10:        if k ∉ knownThreads then                    ▷ first trace of a new thread
11:            if threadSpawningPoints[k] =NIL then abort("missing clone")
12:            E ← E ∪ { (threadSpawningPoints[k], ActiveTreePathsByThread[k][1]) }
13:            knownThreads ← knownThreads ∪ {k}
14:        end if
15:        if last(obsSequence[i]) = "clone" then          ▷ thread creation syscall
16:            if latestClone ≠NIL then abort("ambiguous thread creation : overlap")
17:            latestClone ← last(ActiveTreePathsByThread[k])
18:        end if
19:    end if
20:    if isThreadCreation(obsSequence[i]) then
21:        threadSpawningPoints[k] ← latestClone
22:        latestClone ←NIL
23:    end if
24: end for
```

Algorithm 2: Multithreaded version of the call-tree construction algorithm

New threads are merged on the fly with the main graph when the first trace of a new thread is encountered (line 9). This uses information about invocations to `clone` (line 17). The merging is similar to the *fuse* operator described in section 3.4. As with *fuse*, the tree construction algorithm adopts a conservative merging protocol and raises an exception as soon as multiple thread creations overlap (line 16). The algorithm also checks that any new thread can be associated with a '`clone`' invocation (line 10).

These safeguards are needed to insure that ambiguous traces are always referred to the developers. Interestingly, we never encountered such a situation in the examples we have looked at. This seems to indicate that except for pathologically high workloads, thread creation latencies are short enough for ambiguities to disappear, at least on mono-processor machines.

Figure 25 illustrates the construction of a multithreaded tree when algorithm 2 is applied to the traces of figure 24-b. The figure shows the call-tree after each of the three traces gets processed. It also shows how the algorithm maintains two active paths for each thread (in bold black for thread t1 and bold grey for thread t2).

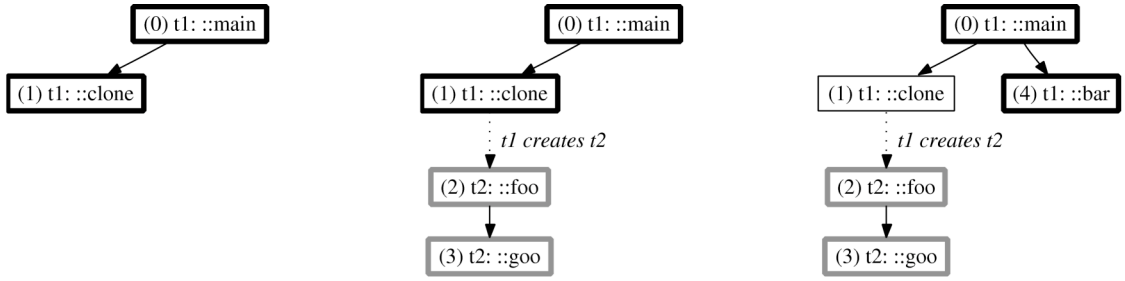
```
main() {
  clone(&foo);
  bar();
}
```

(a) *elementary multithreaded program (simplified)*

```
<traceSet threadCreation="clone">
  <trace thread="1" >
    <call entity="" method="clone" />
    <call entity="" method="main" />
  </trace>
  <new thread="2" />
  <trace thread="2" >
    <call entity="" method="goo" />
    <call entity="" method="foo" />
  </trace>
  <trace thread="1" >
    <call entity="" method="bar" />
    <call entity="" method="main" />
  </trace>
</traceSet>
```

(b) *the trace sequence obtained for program (a)*

Figure 24: A toy multithreaded program and its trace sequence to illustrate algorithm 2



(a) After processing thread t_1's first stack trace. latestClone now points to node *(1) t1: ::clone* (lines 15-18 of Algorithm 2).

(b) After handling the creation of thread t_2 (lines 20-23) and processing t_2's first trace (lines 9-14)

(c) Final call-tree, once the second trace of thread t_1 has been processed

Key: ▭→▭: ActiveTreePathsByThread[1]; ▭→▭: ActiveTreePathsByThread[2]

Figure 25: Step by step illustration of the multithreaded tree construction (algorithm 2) on the toy program of figure 24

4.2 A simple yet powerful graph calculus

One of the key features of CosmOpen's graph manipulation language are *graph variables*. These can be used to combine elementary operators into complex filters. For instance: in the following excerpt a graph is loaded from the file `jtc.xml` into variable `A`, and another from `orbacus.xml` into variable `B`. The contents of `A` is assigned to the variable `C`, and the content of `B` added to `C`—i.e. `C` ← `A` ; `C` ← `C` ∪ `B`. The resulting graph is then saved in file `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 language fall into four categories:

1. **Generic Management Commands** allow users to interact with the interpreter, for instance to set/unset options, run external shell commands, or execute a script from a file.

2. **Input/Output Commands** provide support to save and load graphs, and convert graphs both into postscript and dot format. As explained in section 3.2, some special 'binding' commands provide users with a simple WYSIWYG mechanism when manipulating graphs.
3. **Transformation Commands** constitute the biggest group, and provide operators for variable management (assignment, deletion, etc.), set algebra (union, complement), 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 `forward` computes all the calls directly and indirectly made by the nodes contained in a graph. In section 3, we used `forward` (line 6 in figure 19) to select the function calls that were internal to the `pthread` library.

The main operators for transformation and extension are listed in Table 1 with a short description. A detailed specification of each operator is provided in the appendix. Most of them take a combination of either variables or patterns (explained below) as parameters.

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

Pattern expressions

Patterns are wildcards expressions similar to those found in Unix shells. They allow users to select interesting sets of nodes based on node names. Node names use the following syntax:

```
[<namespace>::][<class>]::<method>'<thread_ID>-<sequence_nb>
```

where `[..]` denotes an optional element, and `<method>` denotes a method or a function. Because node names embed information about classes, methods, thread and sequence numbers, patterns allow developers to filter calls according to a wide range of criteria. For instance the pattern `*'t1-*` selects all invocations performed by the thread `t1`, while `::pthread_*` selects all invocations to functions whose name starts with `pthread`. We used this in section 3 when we removed the calls that were internal to the `pthread` library. Many operations exist both in a variable-only and a pattern-based form. For instance '`add A B`' adds the graph content of `A` to `B`, while '`put ::pthread_* A B`', adds all the nodes that start with `pthread_` in `A` to `B`.

Abstraction

Besides the recursive operators, which use the traditional notion of recursive closure found in graph theory, the construction of high-level models relies on two other key operators: `abstract` and `absPattern`. Their effect is to remove nodes, but keep the connectivity they provided. Both operators reconnect all the children of the removed nodes with the parents thereof. (See the appendix for a precise specification of this operation.) For instance, if a graph contains three nodes connected as x → y → z, abstracting node 'y' away from the graph will result in x → z: node 'y' is gone, but its former child ('z') is now connected to its former parent ('x'). In the example of section 3.5, abstraction is the operation we used to remove the last calls to the threading library.

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), and to scope down the analysis to a particular phase of the program's execution (e.g. after a socket connection has been accepted).

The next step (*seed-based selection*) selects a so-called *seed*, a small set of operations one is interested in, and uses extension commands to compute either the calls made by this subset, or the calls leading to this subset. Depending on the complexity of the resulting call-tree, this seed-extension might be repeated, and the results combined through union and complement.

The last step (*abstraction*) uses abstraction operators (`abstract`, `absPatern`) to remove intermediary classes and functions, and to retain the key program elements involved in the interaction.

This fundamental pattern is followed by the script we presented at the end of section 3 (repeated in figure 26 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 selects the immediate children of this seed. 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 select the calls internal to the `pthread` library, which are then removed from the final graph 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 26: The script we used to abstract the pthread library in section 3

5 Case study: non-determinism in a CORBA ORB

We have applied CosmOpen to three well-known industrial CORBA platforms (ORBacus [28], TAO [29], and omniORB [30]) to analyse how multithreading could interfere with replication mechanisms. We have presented elsewhere [6, 7] our findings for ORBacus. In this section, we report on how we conducted this analysis, with a particular focus on ORBacus. Although this does not constitute a controlled experiment, our aim here is to illustrate the practicality of our tool on a real example.

5.1 CORBA and Fault-Tolerance

CORBA is a standard for communication-oriented middleware developed by the OMG [31]. 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 platform. With CORBA, programs developed in different languages (Java, C++), running on different OS (Windows, Linux, Solaris *etc.*), can be easily "glued" together (figure 27).

Over the years, CORBA integrated many additional technologies, such as distributed events, transactions, real-time, and components. An important effort was made to include fault-tolerance and resulted in the specification of Fault-Tolerant CORBA (FT-CORBA) [32]. FT-CORBA does not cover all needs of distributed fault-tolerant applications, and in particular does not provide 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 kernel-level lock operations could influence the internal request handling of the middleware.

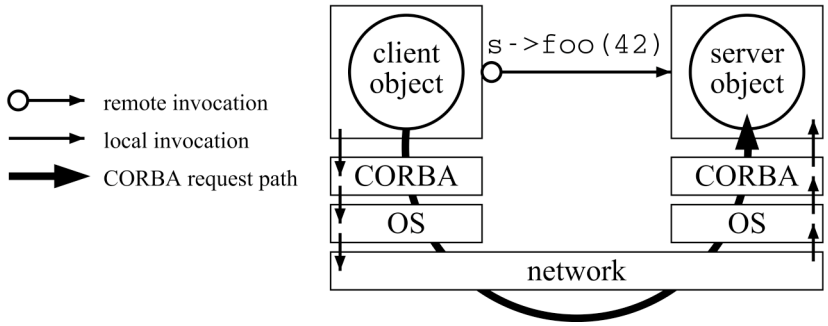


Figure 27: **Remote Method Invocation in CORBA**

This required that we understood precisely how requests progress from the OS network interface (system calls such as `socket`, `recv`, `send`, etc.) to the application (application level interface), while tracking synchronisation activities (essentially mutexes). We also wanted to trace object creations (in C++) 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 implemented a basic CORBA application, and traced the activity of the server while one request was processed and the server terminated. Following the approach that we presented in section 3.2, we identified a set of 59 breakpoints 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 the 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 operations considerably slows down a program's execution. This is most noticeable during a program's 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 spent initialising the ORB, which is not the phase we were interested in. For comparison, a non-monitored run would take less than 1 second. To avoid this intractable observation cost, we applied an adaptive observation approach (section 4.1), and waited until the middleware had been initialised before we automatically activated the most costly breakpoints. This reduced the duration of a monitored run with ORBACUS to 4min 53s (283s), a more than tenfold increase in performance.

The number of threads, traces, items and invocations obtained for ORBACUS, 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, COSMOPEN reconstructed a call-tree containing 2066 invocation nodes (section 4.1).

One rationale behind the use of breakpoints to generate a call-tree is that a single breakpoint activation yields information about all the frames of 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 obtained) 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), but all represent substantial gains over an approach that would not use stacks (from 68% to 213%).

The breakdown of the breakpoint activations leading to these traces is represented on figure 28. 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, only 203 of these 538 mutex operations occur between request reception (return from the system call `recv`) and the corresponding reply being sent (return from the system call `send`). This second number is more representative, as it ignores binding overheads, and corresponds to the critical path of a request within the ORB. Table 3 compares the number of mutex operations on the critical path of the three ORBs (first invocation, not averaged). We see that the synchronisation activity of ORBACUS is significantly higher than in others ORBs, but, and this may come as a surprise, that OMNIORB and TAO also generate an important number of mutex operation themselves.

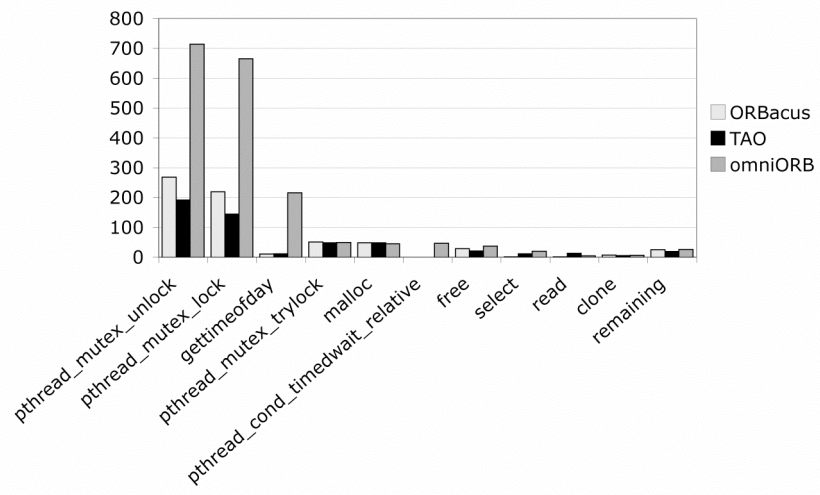


Figure 28: Breakdown of activated breakpoints in ORBACUS, TAO, and OMNIORB

5.3 Extracting high level behavioural patterns with COSMOPEN

To understand the role played by the mutex operations of table 3, we first needed to understand how requests are processed in each ORB. Directly analysing any of the obtained call-trees is in practice intractable, due to their respective size. Figure 29 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 size makes it extremely hard to read, to say nothing of any manual analysis.

To analyse this graph, we used the same approach as for the smaller examples we presented in section 3. We repeatedly applied the three steps of *temporal scoping*, *seed-based selection*, and *abstraction* that we introduced in the previous section. Each incremental step gave us new insights in the structure and behaviour of ORBACUS and helped decide which actions should follow.

In terms of scoping, we first abstracted away the multithreading library (see section 3.3). This only removed 30 invocations, but most importantly, it created the proper thread creation links. We then searched for networking activity (using COSMOPEN's `print` command), and discovered that `recv`, and `send` were the only network primitives used by ORBACUS. 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 commands to get a fuller picture of all invocations in the ORB that would result 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 29, `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
```

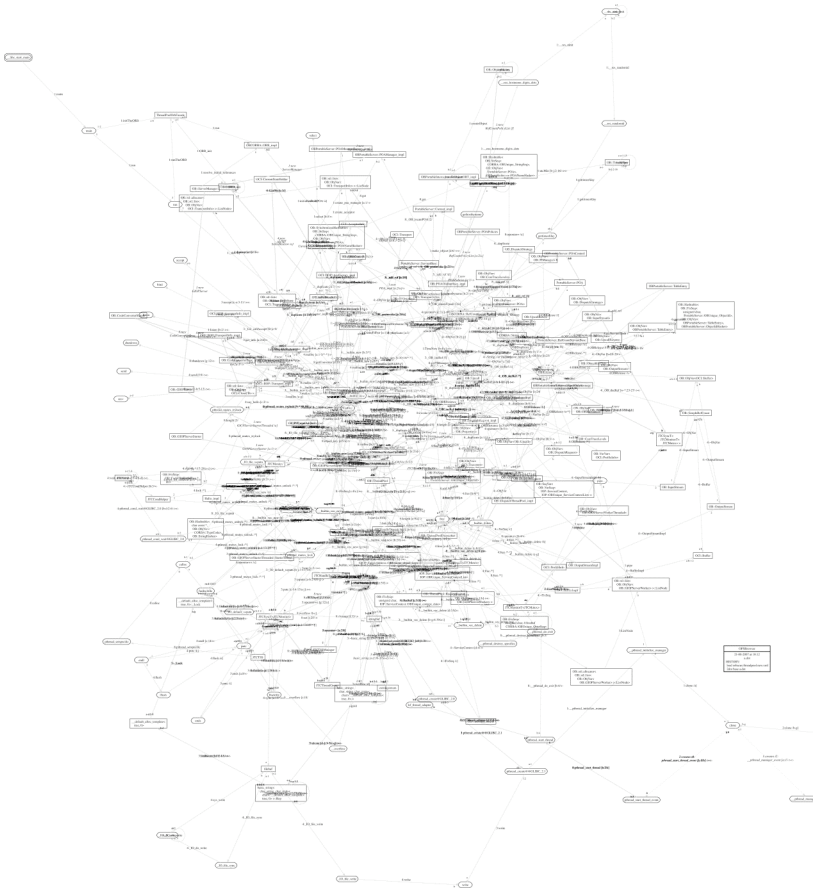


Figure 29: The complete call-tree or ORBACUS reconstructed by COSMOPEN (presented here as a class interaction diagram)

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 `pthread` system library, we removed these libraries from the call-tree (*abstraction*), ending with 27 remaining invocations. We used this graph as a backbone to inspect ORBACUS' code, and we constructed complementary graphs to analyse the activity of various other threads. This led us to identify additional key entities or activities (such as 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
```

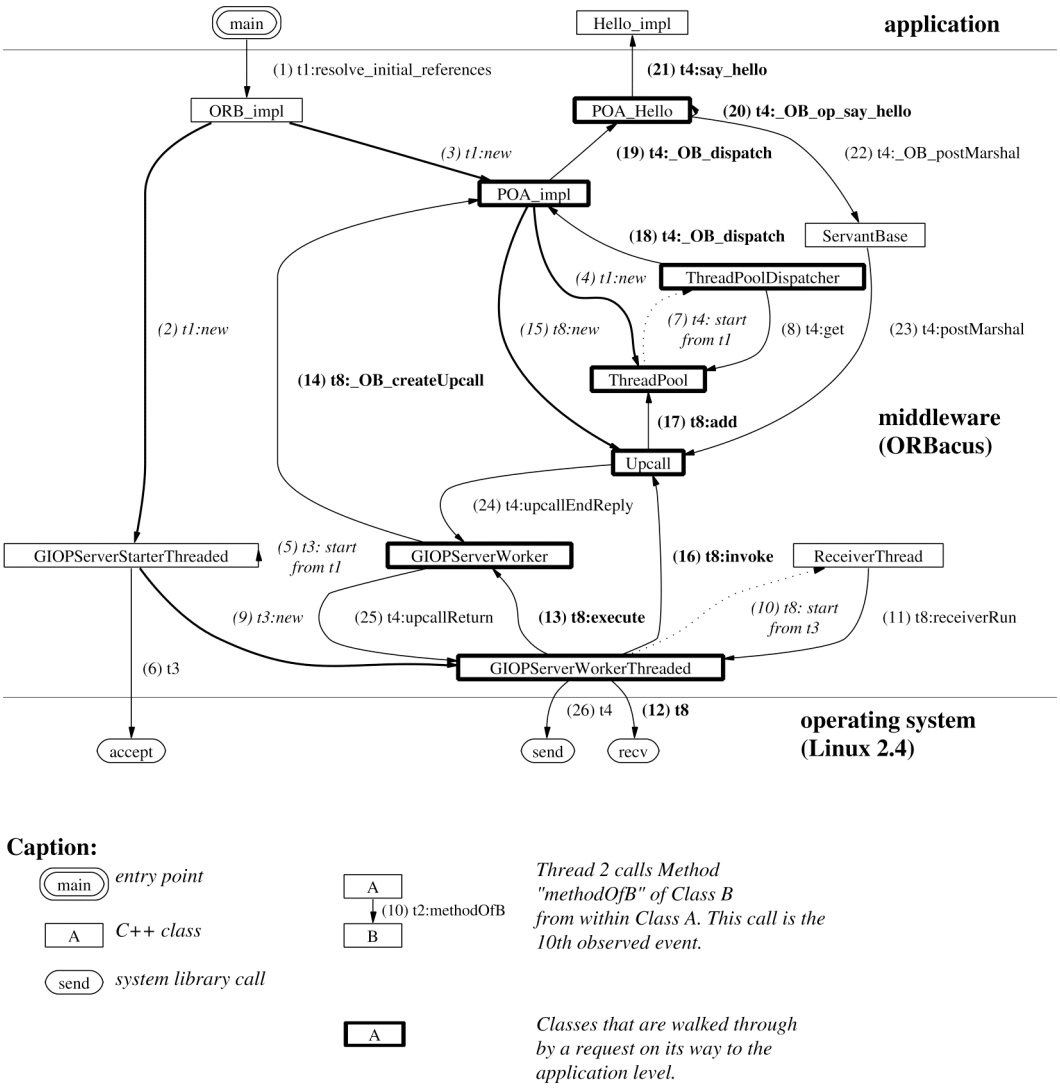


Figure 30: High level representation of the request processing in ORBACUS

The resulting graph contained 36 invocations. We finally removed delegate classes, producing the final graph (figure 30). 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 (t*1*, the main thread, t*3* the thread that accepts connections, t*8* the receiver thread, and t*4* 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 (t*3* is spawned by t*1* in Step *(5)*, t*8* by t*3* in Step *(10)* for instance) and object allocations (a new `ThreadPool` object is allocated by Thread t*1* in Step *(4)*). For clarity reasons, mutex activity is not shown in 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 a large set of behavioural data (figure 29) a higher-level representation of a program's execution. This new representation highlights some key aspects of the platform (here request processing) and this for a very reasonable cost of observation.

5.4 Discussion and limitations

As illustrated by this case study, COSMOPEN provides a reverse engineering environment that is both flexible and cost-effective. This, however, comes with at least three main limitations. First,

CosmOpen strongly depends on an underlying stack capture mechanism. It cannot detect interactions such as in-lined functions that do not appear in the stack. The only exception are activities that can be reconstructed from the ordering of events (such as a pipe IPCs), but as we discussed in section 3.4 this again only works in simple cases showing limited concurrency.

Another limitation regards the ordering of events. The sequence numbers assigned by CosmOpen reflect the order in which invocations are *observed*, but not always the order in which they *occur*. There are, however, some guarantees. Sequence numbers faithfully reflect the order of execution between two calls in three cases: (i) when both calls were made by the same thread; (ii) when they both correspond to breakpoint activations; or (iii) when they can be linked together by repeatedly applying the two previous rules. Developers must be aware of this semantics when using temporal operators such as `fuse`.

Finally, the repeated application of graph operators does not guarantee that the resulting model is consistent. For instance, if calls to a given library (e.g. `pthread`) are removed by abstraction from a given graph, but left in another, then merging the two graphs can create spurious results (in particular if the two original graphs overlap).

The first two limitations are inherent to the use of partial stack-trace information. The last one arises from the use of a flexible (and hence expressive) manipulation language. We are currently investigating how best to mitigate these three limitations. In particular we are considering the use of better feedback mechanisms to prevent users from drawing wrong conclusions or applying inadequate operators.

6 Related work

As mentioned in our problem statement, early reverse-engineering work has essentially focussed on static structures and source code analysis. The past decade has however seen numerous efforts to use dynamic data to reverse engineer software [16, 17, 18, 33, 34, 35, 36, 37, 38]. Stack trace analysis, as used in CosmOpen, has also found some useful applications in particular in the field of high-performance computing, for performance analysis and anomaly detection [39, 40, 41, 42]. Some of these techniques use pure aggregative techniques to reduce the size and complexity of the data such as the spiral timeline presented in [16]. In the following we ignore these purely aggregative techniques, and focus instead on techniques that conserve some form of call-graph information to analyse dynamic program data.

We have already discussed the work of Jerding et al [17] and Pauw et al [18] in section 2. In an early approach related to our work, Richer et al. [34] proposes a domain-specific query language to extract higher-level models from both static and dynamic information. This language, based on Prolog, focuses on the structural notion of "components" obtained through clustering of program entities, something we do not consider here. Instead, we primarily looked at the tension between observation costs and information completeness, and at issues of cross-layer entangling in layered platforms.

In a related approach, *AVID* (*Architectural VIzualisation of Dynamics*) presented by Murphy et al [33, 37] reduces the complexity of dynamic behavioural data by constructing an architectural view of a running object-oriented program. The tool records method invocations, object allocations and de-allocations. To visualise an execution, the user must first provide a mapping of low-level entities (objects) to higher-level groupings (*collections*) that make sense for the task at hand. This grouping occurs off-line and is static. For each collection, the tool counts particular events (such as the number of object allocated) and represents them as histograms attached to the collection. The tool also draws an edge whenever an object in a particular collection invoked an object in another one, and labels this edge by the number of invocations between the two collections. These counters and histograms obey a 'replay' metaphor by which a user can visualise how they evolve over time. The current state of the call stack at the point of visualisation is represented as a

path running through the collections that are traversed by the program's thread (which the authors call an *hyperarc*). Compared to CosmOpen, the grouping of objects used by Walker et al is not dissimilar to our own abstraction operators where a call A → B → C is replaced into A → C, and A gets to represent both A and B.

To reduce the size of the resulting traces, the second version of *AVID* [37] considers various sampling strategies. Interestingly, some of these strategies involve capturing stack snapshots every *x*-events and comparing subsequent snapshots to compute method entries and exits. However, the authors do not consider the reconstruction of a call-tree from these stack traces, and contrary to CosmOpen, do not use well-identified breakpoints to generate their traces. Also *AVID* does not support advanced graph manipulations.

Shimba [35] is a reverse engineering environment for Java that supports the parallel exploration of both static and dynamic views of a program. Behavioural information is represented as extended UML sequence diagrams, and structural information as a use/dependency graph. *Shimba* allows users to correlate structural and behavioural data by filtering one type of data using the other (a technique termed *model slicing*). *Shimba* also offers advanced analysis techniques to reduce the size of dynamic data: (i) it can synthesise statecharts from sequence diagrams; (ii) it can also detect behavioural patterns in sequence diagrams and replace them by a repetition construct.

Trace events are collected in *Shimba* using debugging breakpoints and can be scoped to a subpart of the program. Although *Shimba* does not reconstruct call-trees from stack traces as we do, it uses some of the information contained in the stacks to identify callers of intercepted invocations. The structural analysis part of *Shimba*, *Rigi*, also provides a library of Tcl/Tk scripts to manipulate structural graphs. The supported operations are similar to those of CosmOpen (transitive closure, union) but are limited to static data.

Not dissimilar to *Shimba*, *BLOOM* [36] is an integrated system for software visualisation, covering data collection, analysis, and visualisation of both static and dynamic information. One of its key features is a visual language that allows users to specify what should be represented and how. *BLOOM* works on event traces that contain method invocations, exits, and memory management events (allocation, de-allocation). Among the analysis provided, *BLOOM* can construct direct acyclic graphs from the trace data in which identical call-paths are collapsed together. *BLOOM* also offers a 'package encoding' analysis that allows users to specify how particular library calls should be merged together. Of interest for CosmOpen is the *trace sampling analysis* that samples the event trace at regular intervals while retaining all calls needed for the sampled routines. *BLOOM* also provides a powerful querying language, similar to relational algebra (joins, selects and projections). The final visualisation is obtained by mapping a query to the parameters of one of the visualisation mechanisms (time-line spiral, 3D or 2D scatter maps, graphs, etc.). Compared to CosmOpen, *BLOOM* does not consider the construction of call-trees from partial stack traces. Because of its high expressiveness, *BLOOM*'s visual query language probably supports graph manipulation similar to those of CosmOpen. This comes however at the cost of a higher complexity, as *BLOOM* does not provide a dedicated graph manipulation language.

In [38], Zaidman et al use a web-mining algorithm on execution traces to discover key classes in an object-oriented program. Their approach uses exhaustive tracing on the core part of a program (excluding library), and constructs a *compacted call graph*, that shows interactions between classes rather than objects (as in our interaction diagram). Multiple invocations between two classes are represented as weighted edges (similarly to *AVID* [33]). The authors apply a data-mining algorithm developed for web pages to compute a 'hubiness' value for each class that measures the extent to which a class acts as a hub in the compacted call graph. The authors show that classes with a high hubiness value tend to be key classes in terms of program comprehension.

In a slightly different field, tools for performance analysis and anomaly detection have also looked at execution traces to diagnose performance problems and debug applications [39, 40, 41, 42]. Contrary to CosmOpen they do not address reverse engineering, and usually target high-performance parallel computers. In these systems, the same code is typically executed on a large

number of nodes, and scalability challenges do not arise so much from code complexity, but rather from the large number of nodes generating performance data. As in CosmOpen, these tools must therefore find ways to extract relevant information from large bodies of observation data.

Of particular interest among performance analysis tools, is a technique known as call-path profiling. This technique distinguishes between invocations based on how functions have been reached, and aggregates performance metrics according to the call-path used to invoke a particular function. A whole range of tools using call-path profiling have been proposed, among which *STAT* [41] and *iPath* [42] from the *Paradyn* project, as well as *Expert* from Wolf et al [39, 40].

STAT was developed to assist with the debugging of parallel applications running on extreme-scale supercomputing infrastructures, comprising more than 10^3 processors. *STAT* attacks the scalability challenge of such environments from two angles: i) it reduces the amount of collected data by sampling stack traces rather than using an exhaustive tracing facility; and ii) it uses a tree-based overlay to efficiently aggregate profiling data across 1000s of processors without engorging the debugging front-end. By sampling stack traces, *STAT* is able to form '*process equivalence classes*', i.e. group process into classes that exhibit similar behaviours. This allows *STAT* to considerably reduce the amount of data presented to developers, and works because the processes involved in large parallel applications, although their number might be very large, usually show very strong similarities in their behaviour. Differences between process classes are shown through node colouring. For comparison, the call-tree constructed by *STAT* is a simplified version of the one used by CosmOpen, and does not contain any sequence numbers. Formally, it corresponds to the one CosmOpen would construct without properties P2 (breakpoint discrimination) and P3 (order conservation). This is understandable as *STAT* is neither interested in individual invocations (P2) nor in the relative ordering of call paths and stack traces (P3).

In another *Paradyn* paper [42] Bernat and Miller introduce *iPath*, a tool that uses dynamic code instrumentation to insert and remove performance probes while the program is running. *iPath*'s probes use a stack-walk mechanism to capture stack traces, as we do with `gdb`, and use the call-path information as a context to the various performance metrics being gathered. As with *STAT* no information is gathered on the ordering of individual stack traces and instead all invocation resulting from the same call-path are merged together.

Interestingly in [41] the authors of *STAT* discuss the visual complexity of the call-graphs *STAT* generates, and propose pruning as a possible approach to further reduce the size of the trees presented to users. One of the pruning strategies they suggest (but do not appear to have implemented), is based on library APIs (e.g. the MPI APIs), and consists in cutting all subtrees that start with a library call. Pruning is, in essence, the philosophy we followed in CosmOpen, with the key difference that the graph operators we propose offer much richer possibilities than a simple graph pruning.

Expert, developed by Wolf et al [39, 40], also uses call-path profiling, but rather than focussing on the handling and visualisation of traces and their associated graphs, consider automatic diagnosis. *Expert* automatically searches event traces for patterns that denote typical problematic situations (for instance a receiver waiting for a message). *Expert* uses a repository of known efficiency problems to generate high-level performance diagnostics that are then represented with the *CUBE* visualisation software [43]. To improve scalability, *Expert* uses a *successive refinement* technique [39]. This technique relies on a hierarchy of patterns to quickly home in onto potential problems, before refining the diagnostic with a finer-grained—but more expensive—analysis.

CUBE [43] is the visualisation component used by *Expert*. *CUBE* uses three types of data, or *dimensions*: a metric dimension (what is being measured), a call-tree dimension (where in the code it is being measured, i.e. call-paths), and a system dimension (on which node this is happening). These dimensions are represented graphically as trees, and navigability is provided by allowing users to collapse or expand a particular tree node. *CUBE* also offers an alternative 2D or 3D Cartesian representation to represent the system's topology.

Compared to CosmOpen, *STAT*/*Paradyn*, *Expert* and *CUBE* do not focus on graph manipulation, but rather on how to aggregate metrics and detect performance problems. The use of multi-level patterns to detect anomalies in *Expert* shares the same philosophy as filtering through graph operators in CosmOpen, but on a different kind of data, and for a different aim (diagnosis rather than program comprehension). Furthermore, although these tools extract a dynamic call-graph, this call graph does not distinguish between individual invocations, and thus contains less information that the call-tree reconstructed by CosmOpen. In particular, none of these tools consider the problem of reconstructing a time-stamped call-tree from a sequence of stack traces.

7 Conclusion and perspectives

This paper has introduced CosmOpen, a practice-driven tool for the behavioural analysis of complex multi-layer software. 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. Thanks to its inexpensive event extraction scheme CosmOpen can be applied to large industrial software without instrumentation. Its graph calculus engine implements a novel approach to the navigation and visualisation of large behavioural data that is based on partial collapsing. With CosmOpen, 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, CosmOpen specifically targets multi-level software by allowing its user to adjust the "focal length" of the reverse-engineering process to specific abstraction planes.

The motivating example of section 3 (`pthread`) and the larger case study of section 5 (Orbacus) both illustrate how CosmOpen can help extract rich insights from complex software with no or very little a priori knowledge. CosmOpen only requires that developers be aware of a component's upper and lower interfaces and of their broad semantics. Armed with this only knowledge we were able to reverse engineer the role of the thread manager in the multi-threading library `pthread`, and to discover the key internal classes and the threading strategy employed by Orbacus for request processing.

CosmOpen 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 code browsing facilities, and the explicit management of abstraction planes to prevent inconsistent graph manipulations.

In terms of evaluation, much remains to be done to quantify the help provided by CosmOpen. Our personal experience is that CosmOpen considerably eases the process of analysing an unknown piece of code, compared to source browsing or to manual step-by-step execution. Using controlled experiments to compare CosmOpen with other dynamic reverse-engineering tools should however provide more accurate insights in this area.

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.

Acknowledgement

The authors would like to thank the anonymous reviewers for their detailed and constructive feedback. Their help was instrumental in improving this article. The first author would like to thank Christof Fetzer for his encouragement to publish the work presented here.

8 Bibliography

1. 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.

2. Stolper, SA. Streamlined design approach lands mars pathfinder. *IEEE Software* 1999; **16** (5): 52-62.

3. Ebert J, Kullbach B, Riediger V, Winter A. GUPRO. Generic Understanding of Programs - an overview. *Electronic Notes in Theoretical Computer Science* 2002; **72**(2): 47-56.

4. Chen YFR, Fowler GS, Koutsofios E, Wallach RS. Ciao: a graphical navigator for software and document repositories. *Proc. of the Int. Conf. on Software Maintenance*, Opio (Nice), France, 1995; 66-75.

5. Frohlich M, Werner M. Demonstration of the interactive graph visualization system davinci. *Proc. of the DIMACS Workshop on Graph Drawing'94*, Lecture Notes in Computer Science, Springer 1994; **894**: 266-269.

6. Taïani F, Fabre J-C, Killijian M-O. Towards implementing multi-layer reflection for fault-tolerance. *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN-2003)*, San Francisco, CA, IEEE Comp. Soc., 2003; 435-444.

7. Taïani F, Fabre J-C, Killijian M-O. A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures. *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN'2005)*, Yokohama, Japan, IEEE Comp. Soc., 2005; 270-279.

8. Killijian M-O, Fabre J-C. Implementing a reflective fault-tolerance CORBA system. *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, Nürnberg, Germany, 2000; 154-163.

9. Pérennou T, Fabre J-C. A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach. *IEEE Trans. on Computer, Special Issue on Dependability of Computing Systems*, 1998; **47**(1):78-95.

10. Chen YFR, Gansner ER, and Koutsofios E. A C++ data model supporting reachability analysis and dead code detection. *SIGSOFT Softw. Eng. Notes* 1997; **22**(6):414-431.

11. Gansner ER, North SC. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience (SPE)* 2000; **30**(11):1203-1233.

12. Wong K, Tilley SR, Müller HA, Storey M-A. Structural redocumentation: A case study. *IEEE Software* 1995; **12**(1):46-54.

13. Wong K. *The Rigi User's Manual - Version 5.4.4.* Department of Computer Science / University of Victoria: Victoria, BC, Canada, 1998.

14. Graham S, Kessler P, McKusick M. Execution profiler for modular programs. *Software - Practice and Experience (SPE)* 1983; **13**(8):671-685.

15. Ottogalli F-G, Labbé C, Olive V, de Oliveira Stein B, Chassin de Kergommeaux J, Vincent JM. Visualisation of distributed applications for performance debugging. *Proc. of the Int. Conf. in Computational Science (ICCS'01)*, Lecture Notes in Computer Science, Springer, 2001; **2074**:831-840.

16. Reiss SP, Renieris M. Generating Java trace data. *Proc. of the ACM 2000 Conf. on Java Grande*, San Francisco, CA, ACM Press, 2000; 71-77.

17. Jerding DF, Stasko JT, and Ball T. Visualizing interactions in program executions. *Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97)*, ACM Press, 1997; 360-370.

18. De Pauw W, Sevitsky G. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 2000; **12**(14):1431-1454.

19. O'Byrne J. Sharper eyes on the sky. *Sky & Space Magazine* December 1996; 20-24.

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

21. Tatsubori M, Chiba S, Killijian M-O, Itano K. OpenJava: A Class-Based Macro System for Java. *Reflection and Software Engineering*, Lecture Notes in Computer Science, Springer, 2000; **1826**:117-133.

22. Chiba S, Nishizawa M. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, *Proc. of 2nd Int. Conf. on Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science, Springer, 2003; **2830**:364-376.

23. Schwarz B, Debray S, Andrews G. Disassembly of Executable Code Revisited. *Proc. of the 2002 IEEE Working Conf. on Reverse Engineering (WCRE 2002)*, Richmond, VA, 2002; 45-54.

24. Stallman R, Pesch RH. *Debugging with GDB*. The Free Software Foundation: Boston, MA, USA, 9th edition, 2002.

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

26. Linus Torvalds, *Re: SCO: "thread creation is about a thousand times faster than*, e-mail sent 27 Aug 2000 22:38:24 -0700 and surrounding discussion. http://lkml.org/lkml/2000/8/28/97 [24 September 2008]

27. Sun Microsystems JVM Tool Interface (JVMTI), Version 1.0, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, [24 September 2008]

28. Orbacus Documentation, http://www.orbacus.com/support/docs.htm [30 October 2008]

29. Schmidt D, Cleeland C. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine, Special Issue on Design Patterns,* 1999; **16**(4): 54-63. DOI: 10.1109/35.755450

30. Grisby D, Lo S-L, Riddoch D. The omniorb version 4.0 user's guide. http://omniorb.sourceforge.net/omni40/omniORB/ [24 September 2008]

31. *Common Object Request Broker Architecture: Core Specification (Version 3.0.2 - formal/02-12-02)*. OMG: Needham, MA, U.S.A., 2002.

32. Fault Tolerant CORBA, Chapter 23 of *Common Object Request Broker Architecture: Core Specification (Version 3.0.2 - formal/02-12-02)* [31], 2002.

33. Walker RJ, Murphy GC, Freeman-Benson B, Wright D, Swanson D, Isaak J. Visualizing Dynamic Software System Information through High-Level Models. *Proc. of the 13th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, ACM Press, 1998; 271-283.

35. Systä T, Koskimies K, Müller H. Shimba – an environment for reverse engineering Java software systems. *Software Practice and Experience (SPE)*, 2001; **31**(4):371-394. DOI: 10.1002/spe.386

36. Reiss SP, Renieris M. The BLOOM Software Visualization System. in *Software Visualization: from Theory to Practice*, K. Zhang, Ed. Kluwer Academic Publishers, 2003.

37. Chan A, Holmes R, Murphy GC, Ying AT. Scaling an Object-Oriented System Execution Visualizer through Sampling. *Proc. of the 11th IEEE Int. Workshop on Program Comprehension* (IWPC'03). IEEE Computer Society, 2003; 237-244. DOI: 10.1109/WPC.2003.1199207

38. Zaidman A, Calders T, Demeyer S, Paredaens J. Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process. *In Proc. of the Ninth European Conf. on Software Maintenance and Reengineering (CSMR'2005)*. IEEE Computer Society, 2005; 134-142. DOI: 10.1109/CSMR.2005.12

39. Wolf F, Mohr B, Dongarra J, Moore S: Efficient Pattern Search in Large Traces through Successive Refinement. *Proc. of the 2004 European Conf. on Parallel Computing (Euro-Par)*, Lecture Notes in Computer Science, Springer, 2004; **3149**:47-54.

40. Becker D, Wolf F, Frings W, Geimer M, Wylie BJN, Mohr B. Automatic Trace-Based Performance Analysis of Metacomputing Applications. *Proc. of the IEEE Int. Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, 2007. DOI: 10.1109/IPDPS.2007.370238

41. Arnold DC, Ahn DH, de Supinski BR, Lee GL, Miller BP, Schulz M. Stack Trace Analysis for Large Scale Debugging. *Proc. of the IEEE Int. Parallel & Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, 2007. DOI: 10.1109/IPDPS.2007.370254

42. Bernat AR, Miller BP. Incremental Call-Path Profiling. *Concurrency: Practice and Experience* 2007; **19**(11):1533-1547.

43. Geimer M, Kuhlmann B, Pulatova F, Wolf F, Wylie BJN. Scalable Collation and Presentation of Call-Path Profile Data with CUBE. *Proc. of the Int. Conf. on Parallel Computing : Architectures, Algorithms and Applications (ParCo 2007),* NIC series, John von Neumann Institute for Computing, Jülich/Aachen, Germany, 2007; **38**:645-652.

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

45. François Taïani. From stack traces to call-trees: outline of a proof. Technical Report COMP-010-2009, Computing Department, Lancaster University, UK, January 2009.

9 Tables

Table 1: The main graph operators of CosmOpen (Overview)

COSMOPEN provides a default graph variable, `base`, that contains all known nodes and edges. Certain commands use `base` as a default value for optional parameters. These optional parameters are indicated with square brackets `[.]`. Please refer to the appendix for a detailed specification of each operator.

Variable management

`clear G`	removes all nodes from graph variable `G`
`delete G`	deletes the graph variable `G`
`assign G H`	assigns the graph contained in `H` to `G`
`print G`	prints the nodes of graph `G`
`print p G`	prints the nodes of graph `G` that match pattern `p`.

Set algebra operators

`add G H`	adds to `H` the graph contained in `G`
`exclude G H`	excludes the nodes of graph `G` from `H`
`abstract G H`	abstracts away the nodes of graph `G` from graph `H`

Pattern based operators

`put p [G] H`	adds to `H` the subgraph of `G` that matches pattern `p`
`remove p G`	removes the nodes of `G` that match pattern `p`
`absPattern p G`	abstracts away the nodes of `G` that match pattern `p`

Temporal operators

`remAfter n G`	removes the nodes of `G` whose timestamp is greater than `n`
`remBefore n G`	removes the nodes of `G` whose timestamp is less than `n`
`slice n-m [G] H`	adds to `H` the subgraph of `G` with timestamps between `n` and `m`
`fuse p q G`	recreates child-parent relationships based on temporal sequences

Recursive extension operators

`backward G [H]`	computes the backward closure of graph `G` within graph `H`
`forward G [H]`	computes the forward closure of graph `G` within graph `H`
`spread G [H]`	union of the forward and backward closures of a graph
`backN n G [H]`	computes the `n`-step backward extension of graph `G` within graph `H`
`forwN n G [H]`	computes the `n`-step forward extension of graph `G` within graph `H`
`spreadN n G [H]`	union of the `n`-step forward and backward extensions of a graph
`envlop G [H]`	computes the envelop of graph `G` within `H`

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

Table 3: Lock operations during request handling in popular ORBs (first invocation)

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

10 Appendix: specification of the operators of CosmOpen

10.1 Abstract graph operations

In the following, graphs are directed and defined as a pair (V, E) of a vertex set V and an edge set $\mathsf{E} \subseteq \mathsf{V}\times\mathsf{V}$. Sets are noted with sans-serif uppercase letters: X, A; individual nodes with lowercase letters: x, y; and graphs either as a pair of sets: (V, E), or with bold uppercase letters: $\mathbf{H}$, $\mathbf{G}$.

The *union* of two graphs (V, E) and (W, F) is defined as:

$$(\mathsf{V}, \mathsf{E}) \cup_g (\mathsf{W}, \mathsf{F}) \equiv (\mathsf{V} \cup \mathsf{W}, \mathsf{E} \cup \mathsf{F})$$

The *restriction* of a graph (V, E) to a set of nodes A (also termed the graph *induced* by A):

$$(\mathsf{V}, \mathsf{E})_{|\mathsf{A}} \equiv (\mathsf{V} \cap \mathsf{A}, \mathsf{E}_{|\mathsf{V}\cap\mathsf{A}})$$

where $^{\mathsf{E}}|_{\mathsf{X}}$ denotes the restriction of the relation E to the domain X, i.e.

$$\mathsf{E}_{|\mathsf{X}} \equiv \mathsf{E} \cap (\mathsf{X} \times \mathsf{X})$$

The *removal* of a set of nodes A from a graph (V, E) is defined as:

$$(\mathsf{V}, \mathsf{E}) \setminus_g \mathsf{A} \equiv (\mathsf{V}, \mathsf{E})_{|\mathsf{V}\setminus\mathsf{A}} = (\mathsf{V} \setminus \mathsf{A}, \mathsf{E}_{|\mathsf{V}\setminus\mathsf{A}})$$

The *envelop* of a graph (V, E) within another (W, F) is defined as:

$$(\mathsf{V}, \mathsf{E}) \uparrow (\mathsf{W}, \mathsf{F}) \equiv (\mathsf{V}, \mathsf{E} \cup \mathsf{F}_{|\mathsf{V}}) = (\mathsf{V}, \mathsf{E}) \cup_g (\mathsf{W}, \mathsf{F})_{|\mathsf{V}}$$

The *one-step forward extension of a set of nodes* A within a graph (V, E) is defined as:

$$\mathsf{A} \triangleright (\mathsf{V}, \mathsf{E}) \equiv \left(\mathsf{A} \cup \left\{ y \in \mathsf{V} \,\middle|\, \exists\, x \in \mathsf{A} : (x, y) \in \mathsf{E} \right\}, \left\{ (x, y) \in \mathsf{E} \,\middle|\, x \in \mathsf{A} \right\}\right)$$

The *one-step forward extension of a graph* (A,F) within another graph (V,E) is defined as:

$$(\mathsf{A}, \mathsf{F}) \triangleright^1_g (\mathsf{V}, \mathsf{E}) \equiv (\mathsf{A}, \mathsf{F}) \cup_g (\mathsf{A} \triangleright (\mathsf{V}, \mathsf{E}))$$

The *n-step forward extension* of a graph $\mathbf{G}$ within another graph $\mathbf{H}$ is defined as:

$$\mathbf{G} \triangleright^0_g \mathbf{H} \equiv \mathbf{G}$$

$$\mathbf{G} \triangleright^n_g \mathbf{H} \equiv (\mathbf{G} \triangleright^{n-1}_g \mathbf{H}) \triangleright^1_g \mathbf{H}$$

Finally *the forward closure* of a graph $\mathbf{G}$ within another graph $\mathbf{H}$ is defined as:

$$\mathbf{G} \triangleright^\infty_g \mathbf{H} \equiv \bigcup_{i=0}^{\infty} \mathbf{G} \triangleright^i_g \mathbf{H}$$

Similarly we define the *one-step backward extension of a set of nodes* A within a graph (V, E) as:

$$\mathsf{A} \triangleleft (\mathsf{V}, \mathsf{E}) \equiv \left(\mathsf{A} \cup \left\{ y \in \mathsf{V} \,\middle|\, \exists\, x \in \mathsf{A} : (y, x) \in \mathsf{E} \right\}, \left\{ (y, x) \in \mathsf{E} \,\middle|\, x \in \mathsf{A} \right\}\right)$$

The *one-step backward extension of a graph* (A,F) within another graph (V, E) is defined as:

$$(\mathsf{A}, \mathsf{F}) \triangleleft^1_g (\mathsf{V}, \mathsf{E}) \equiv (\mathsf{A}, \mathsf{F}) \cup_g (\mathsf{A} \triangleleft (\mathsf{V}, \mathsf{E}))$$

The *n-step backward extension* of a graph $\mathbf{G}$ within another graph $\mathbf{H}$ is defined as:

$$\mathbf{G} \triangleleft^0_g \mathbf{H} \equiv \mathbf{G}$$

$$\mathbf{G} \triangleleft^n_g \mathbf{H} \equiv (\mathbf{G} \triangleleft^{n-1}_g \mathbf{H}) \triangleleft^1_g \mathbf{H}$$

And *the backward closure* of a graph **G** within another graph **H** is defined as:

$$\mathbf{G} \lhd_g^{\infty} \mathbf{H} \equiv \bigcup_{i=0}^{\infty} \mathbf{G} \lhd_g^{i} \mathbf{H}$$

The *abstraction* of a set of nodes A from a graph (V, E) is defined as:

$$(\mathsf{V},\mathsf{E}) \circledast \mathsf{A} \equiv \left(\begin{array}{l} \mathsf{V} \setminus \mathsf{A}, \\ \mathsf{E}_{|\mathsf{V}\setminus\mathsf{A}} \cup \left\{ (x,y) \in (\mathsf{V}\setminus\mathsf{A})^2 \;\middle|\; \exists\, n_1, n_2, .., n_k \in \mathsf{A} : (x,n_1), (n_i, n_{i+1}), (n_k, y) \in \mathsf{E} \right\} \end{array} \right)$$

Some operators use the ordering of nodes according to their sequence numbers. If *P(t)* is a predicate on sequence numbers, and V a set of nodes, we define the *scoping* of V to *P(t)* as:

$$\mathsf{V}_{\downarrow P(t)} \equiv \{ x \in \mathsf{V} \mid P(timestamp(x)) = \mathsf{true} \}$$

If V is a set of nodes and $\mathtt{p}$ a pattern (as defined in section 4.2), we note $\mathsf{V}\langle \mathtt{p} \rangle$ the *set of nodes in* V *that fulfil pattern* $\mathtt{p}$. We note $\overrightarrow{\mathsf{V}\langle \mathtt{p} \rangle}$ the *vector of nodes in* $\mathsf{V}\langle \mathtt{p} \rangle$ *ordered according to their sequence numbers* (lowest sequence number first). We note $\overrightarrow{\mathsf{V}\langle \mathtt{p} \rangle}[i]$ the i[th] element of this vector.

The *fusion* of a pair of node sequences ($(a_1,..,a_n)$, $(b_1,..,b_n)$) within a graph (V, E) is defined as:

$$(\mathsf{V},\mathsf{E}) \curvearrowleft \Big((a), (b) \Big) \equiv \Big(\mathsf{V}, \mathsf{E} \setminus \Big\{ (x,b) \in \mathsf{E} \Big\} \cup \Big\{ (a,b) \Big\} \Big)$$

$$(\mathsf{V},\mathsf{E}) \curvearrowleft \Big((a_1,..,a_n), (b_1,..,b_n) \Big) \equiv \Big((\mathsf{V},\mathsf{E}) \curvearrowleft \Big((a_1,..,a_{n-1}), (b_1,..,b_{n-1}) \Big) \Big) \curvearrowleft \Big((a_n), (b_n) \Big)$$

If (V,E) is a graph and $\mathtt{p}$ and $\mathtt{q}$ are patterns, we define the function *fuse()* as:

$$fuse\Big((\mathsf{V},\mathsf{E}), \mathtt{p}, \mathtt{q} \Big) \equiv (\mathsf{V},\mathsf{E}) \curvearrowleft \Big(\overrightarrow{\mathsf{V}\langle \mathtt{p} \rangle}, \overrightarrow{\mathsf{V}\langle \mathtt{q} \rangle} \Big)$$

which is defined on the following domain:

$$\begin{array}{ll} i) & |\mathsf{V}\langle \mathtt{p} \rangle| = |\mathsf{V}\langle \mathtt{q} \rangle| \\ ii) & \forall i : timestamp\left(\overrightarrow{\mathsf{V}\langle \mathtt{p} \rangle}[i] \right) < timestamp\left(\overrightarrow{\mathsf{V}\langle \mathtt{q} \rangle}[i] \right) \\ iii) & \forall i > 1 : timestamp\left(\overrightarrow{\mathsf{V}\langle \mathtt{q} \rangle}[i-1] \right) < timestamp\left(\overrightarrow{\mathsf{V}\langle \mathtt{p} \rangle}[i] \right) \end{array}$$

10.2 Specification of CosmOpen's main operators

Parameters in square brackets `[ ]` are optional and replaced by the default graph variable `base` when absent. `base` contains all the nodes and edges known to COSMOPEN (its universe).

Variable management

`assign G H`	$\mathtt{G} \leftarrow \mathtt{H}$	assigns the graph contained in `H` to `G`
`clear G`	$\mathtt{G} \leftarrow (\emptyset, \emptyset)$	removes all nodes from graph variable `G`
`delete G`		deletes the graph variable `G`
`print G`		prints the nodes of graph `G`
`print p G`	$\mathtt{print\ G}\langle \mathtt{p} \rangle$	prints the nodes of graph `G` that match pattern `p`

Set algebra operators

`add G H`	$\mathtt{H} \leftarrow \mathtt{H} \cup_g \mathtt{G}$	adds to `H` the graph contained in `G`
`exclude G H`	$\mathtt{H} \leftarrow \mathtt{H} \setminus_g nodes(\mathtt{G})$	excludes the nodes of graph `G` from `H`
`abstract G H`	$\mathtt{H} \leftarrow \mathtt{H} \divideontimes nodes(\mathtt{G})$	abstracts away the nodes of graph `G` from graph `H`

Pattern based operators

`put p [G] H`	$\mathtt{H} \leftarrow \mathtt{H} \cup_g \mathtt{G}\vert_{nodes(\mathtt{G})\langle \mathtt{p} \rangle}$	adds to `H` the subgraph of `G` that matches pattern `p`
`remove p G`	$\mathtt{G} \leftarrow \mathtt{G} \setminus_g nodes(\mathtt{G})\langle \mathtt{p} \rangle$	removes the nodes of `G` that match pattern `p`
`absPattern p G`	$\mathtt{G} \leftarrow \mathtt{G} \divideontimes nodes(\mathtt{G})\langle \mathtt{p} \rangle$	abstracts away the nodes of `G` that match pattern `p`

Temporal operators

`remAfter n G`	$\mathtt{G} \leftarrow \mathtt{G} \setminus_g nodes(\mathtt{G})_{\downarrow t > \mathtt{n}}$	removes nodes whose timestamp is greater than `n`
`remBefore n G`	$\mathtt{G} \leftarrow \mathtt{G} \setminus_g nodes(\mathtt{G})_{\downarrow t < \mathtt{n}}$	removes nodes whose timestamp is less than `n`
`slice n-m [G] H`	$\mathtt{H} \leftarrow \mathtt{H} \cup_g \mathtt{G}\vert_{nodes(\mathtt{G})_{\downarrow \mathtt{n} \leq t \leq \mathtt{m}}}$	adds the subgraph of `G` between timestamps `n` & `m`
`fuse p q G`	$\mathtt{G} \leftarrow fuse(\mathtt{G}, \mathtt{p}, \mathtt{q})$	recreates edges based on temporal sequences

Recursive extension operators

`backward G [H]`	$\mathtt{G} \leftarrow \mathtt{G} \lhd_g^{\infty} \mathtt{H}$	computes the backward closure of `G` within `H`
`forward G [H]`	$\mathtt{G} \leftarrow \mathtt{G} \rhd_g^{\infty} \mathtt{H}$	computes the forward closure of `G` within `H`
`spread G [H]`	$\mathtt{G} \leftarrow (\mathtt{G} \lhd_g^{\infty} \mathtt{H}) \cup_g (\mathtt{G} \rhd_g^{\infty} \mathtt{H})$	union of the forward and backward closures
`backN n G [H]`	$\mathtt{G} \leftarrow \mathtt{G} \lhd_g^{\mathtt{n}} \mathtt{H}$	`n`-step backward extension of `G` within `H`
`forwN n G [H]`	$\mathtt{G} \leftarrow \mathtt{G} \rhd_g^{\mathtt{n}} \mathtt{H}$	`n`-step forward extension of `G` within `H`
`spreadN n G [H]`	$\mathtt{G} \leftarrow (\mathtt{G} \lhd_g^{\mathtt{n}} \mathtt{H}) \cup_g (\mathtt{G} \rhd_g^{\mathtt{n}} \mathtt{H})$	union of the `n`-step forward and backward extensions
`envlop G [H]`	$\mathtt{G} \leftarrow \mathtt{G} \uparrow \mathtt{H}$	computes the envelop of `G` within `H`