Learning Communication Patterns in Singularity

Paul Barham, Rebecca Isaacs, Richard Mortier and Tim Harris
{pbar,risaacs,mort,tharris}@microsoft.com
Microsoft Research, Cambridge, UK.

1 Introduction

Modern software is so complicated that it is often infeasible to get a good understanding of a system's dynamic behaviour simply from its source code. Commodity operating systems are a good example: they comprise numerous separately-authored components, large numbers of interacting threads, and extensibility mechanisms that allow new components to be plugged in based on boot-time or run-time configuration settings. Ideally it should be possible to understand this kind of complex system by capturing *dynamic traces* of its behaviour and then applying *machine learning* techniques to these traces to elucidate the structure present in them. In practice, this is extremely difficult because such systems are usually poorly specified and structured, and component interactions are largely unconstrained [2].

In order to sidestep these issues, this work is based on the Singularity research operating system [5] in which all component interactions are well-specified and statically verified. In Singularity, inter-process communication (IPC) is performed over type-safe *message channels*. Each channel is statically checked to conform to a *channel contract* that defines named message types that the channel can carry, and a finite state machine (FSM) whose edges define the message send/receive operations that are valid in a given state.

This environment is extremely suitable for the application of machine learning techniques to understand runtime behaviour. Component communications conform to a statically verified FSM that the OS tracks at runtime, we can be sure that we track all forms of IPC, and we can present results using the names of channel contracts and message types. Section 2 describes IPC in Singularity in more detail.

To explore the potential of applying machine learning in this more structured context we have developed a series of preliminary techniques for capturing the runtime dependencies and dominant interaction patterns between components in Singularity. We have explored three techniques that provide progressively more detailed information:

- Firstly, we capture dynamic traces of the messages that an individual thread sends and receives over a single channel (Section 3). This lets us build a probabilistic finite state machine (PFSM) for the channel, capturing how frequently each edge is followed in a given run. The PFSM can help with code coverage testing and performance tuning.
- Secondly, when a thread communicates over multiple channels, we combine the individual state machines to form a joint-PFSM (Section 4). This lets us learn about some of the relationships between different processes – for instance, a file-server thread receives a "request" message on one channel, then performs series of disk I/O operations over a second channel, before responding to the request.
- Finally, we use our traces as input to the ALERGIA grammar inference algorithm (Section 5). This lets us infer multi-channel PFSMs that capture message sequencing patterns more clearly by allowing multiple nodes to represent the same combination of channel states. For instance, in the file-server example, this lets us observe how frequently the file-server thread is able to respond to a request *without* requiring any disk I/O.

Ultimately we would like to use this information about runtime interactions as a basis for learning performance models of complex systems. Such models are useful for performance prediction, self-management and anomaly detection.

2 Channel contracts

Communication between processes in Singularity occurs only by sending messages on bidirectional *channels*, which are strongly-typed accord-

```
contract TestContract {
    in message A(int x);
    in message B(int x);
    in message C(int x);

    out message X(int x);
    out message Y(int x);
    out message Z(int x);

    state Start: one {
        A? -> X! -> Start;
        B? -> Y! -> Start;
        C? -> Choice;
    }

    state Choice: one {
        Y! -> Start;
        Z! -> Start;
    }
}
```

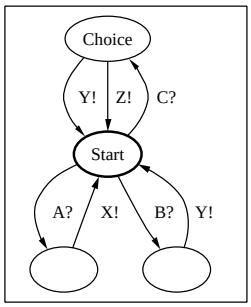

Figure 1: Example channel contract written in Sing#. From the initial state Start, receipt of an A or B message causes a transition to an unnamed intermediate state, from which an X or Y message respectively is returned, before moving back to the Start state. Receipt of a C message moves the channel endpoint to state Choice, from which either a Y or a Z type message is sent. The symbol "?" indicates message receive, and "!" message send. The contract is expressed from the point of view of the exporter (i.e. the server), which must be prepared to receive the **in** messages and to transmit the **out** messages. Conversely, the importer (client), sends the **in** messages and receives the **out** messages.

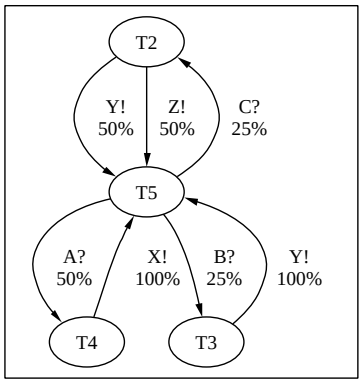

Figure 2: Probabilistic state machine for the Test Contract observed under a synthetic workload. The PFSM demonstrates that, from the start state (`T5`), the `A!` message is twice as likely as either `B!` or `C!`, and that the two replies to the `C!` message are equally likely. Note that the state numbers are arbitrary, assigned by the Sing# compiler.

ing to a *contract*. The Sing# language is an extension of C# that adds support for Singularity communication primitives such as channels and contracts. Channel contracts are defined as finite state machines, specifying the message types and sequence orderings permitted for each type of channel. Compiler verification ensures that contracts are adhered to at runtime. Figure 1 contains the declaration of a simple channel contract, written in Sing#, and a diagram of the equivalent state machine. The channel state machine is expressed from the point of view of the contract exporter (i.e. the server).

Compilation of the example contract produces the classes `TestContract.Imp` and `TestContract.Exp` implementing the client and server endpoints of the channel and supporting methods such as `ep.SendA()` and `ep.RecvZ()`. Message send is an asynchronous operation while receive is synchronous, and messages are guaranteed to be delivered in order.

Channel endpoints are *owned* by one thread at a time, but can be sent inside messages to other processes, thus enabling ownership to be transferred from one thread to another. This transfer is also statically verified.

We observe the inter-process communications by using Singularity's built-in event tracing to log every message as it is sent and received by the respective endpoints. These events record the timestamp, contract type, message type, process id, thread id, endpoint id and contract state.

3 Channel PFSMs

The contract for a channel specifies all possible states and transitions, but doesn't say anything about their expected likelihood. In practice, not all states will be observed equally often, and indeed, different processes may exercise the various parts of a contract with different probabilities. Therefore the probabilistic FSM of a channel contract is a useful augmentation of the static FSM declaration. In particular, the state machine observed at runtime may contain a much smaller number of states and messages than its defining contract. This is valuable code coverage information that can inform testing and performance optimisation. We could also annotate the PFSM with other information such as performance measurements, and this might be useful for performance debugging.

In Singularity, learning the PFSM is rendered straightforward by having an upfront declaration of the state machine. It is then simply a matter of counting the frequency that each state transition occurs. Figure 2 shows the runtime behaviour of the test contract when it was exercised using a very simple test program in which the client sent 1000 messages.

Although this simple annotation of the statically declared state machine is useful for certain tasks, it is not expressive enough to capture critical information about system behaviour. For ex-

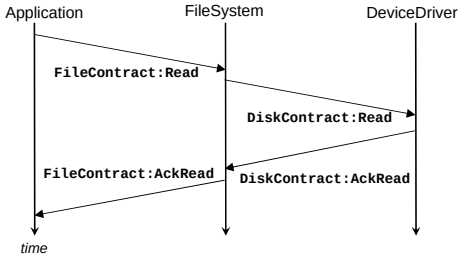

Figure 3: The nesting of the FileContract and DiskContract state machines.

ample, the ordering of state transitions is omitted from the model. This suggests that a richer model is required.

4 Joint PFSMs

Because all inter-process communication in Singularity occurs by sending messages on channels, it is normal for a single thread to simultaneously own more than one channel endpoint. Since there are in fact dependencies between the state machines, the messages of these contracts are usually not arbitrarily interleaved. Crucial to understanding these system dependencies is understanding which interleavings actually occur in practice. For example, a file system thread will typically act as a server in the File Contract, receiving file access requests, and a client in the Disk Contract, issuing disk access requests. Figure 3 depicts this situation.

By participating in multiple channel state machines, each thread instantiates its own "joint" state machine. Every joint FSM provides evidence for causal relationships among its constituent contract types. Unlike the channel FSMs, these joint FSMs can not be declared explicitly, and so their structure, as well as transition probabilities, must be inferred.

To learn the joint probabilistic FSM, we model thread activity using a larger state machine whose states are tuples of the endpoint-states of each endpoint owned by that thread, and whose actions are the union of all messages from their contracts. For example, consider the thread that participates in both the File and Disk contracts as shown in Figure 3. The thread receives read and write requests on a FileSystem contract (with states S_{fs} and messages M_{fs}) and services these requests by making invocations on the DiskDevice contract (S_{disk}, M_{disk}). The behaviour of this thread can be captured using a state space of tuples from $S_{fs} \times S_{disk}$ and edges chosen from $M_{fs} \cup M_{disk}$.

Joint PFSMs enable us to identify quite com-

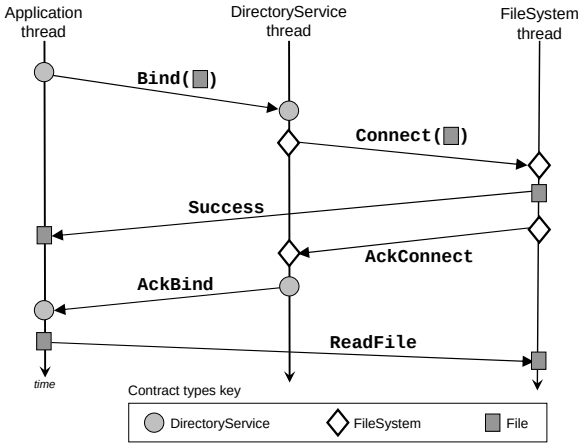

Figure 4: Three state machines inter-depend when an application uses the Directory Service to find and connect to the Filesystem service. This diagram shows how the threads of the application and the two services participate over time in the state machines. Note that the Success and AckConnect messages may be sent by the Filesystem thread in either order.

plex interactions between the various services. To explore this idea in practice, we use a simple file system performance benchmark program called `WebFiles` that selects and reads files from Singularity's port of the SPECweb99 benchmark[5]. Figure 4 describes one of the service interdependencies that occurs in `WebFiles`, a triangular dependency between state machines for application, Directory Service (DS) and Filesystem. The diagram shows two key points: firstly that it is often the case that a transition in one state machine (for example the DirectoryService:AckBind message), cannot occur until a transition occurs in a completely different state machine (the FileSystem:AckConnect). Secondly, the diagram illustrates a channel endpoint being sent to a different thread inside a message (DirectoryService:Bind sends a FileSystem endpoint). This is an instance of a causal relationship between state machines that can be difficult to identify and understand in a running system without access to source code.

An even more complicated scenario appears inside the Filesystem process, which uses a pool of worker threads. When a file request arrives, the main thread pool control thread accepts the request and then hands it off, along with the application's channel endpoint, to one of the worker threads for processing. Figure 5 contains the PFSM that results for one particular worker thread, along with the FSMs for the consitutent contracts. Note that the file read request from the application does not appear in the joint FSM, as

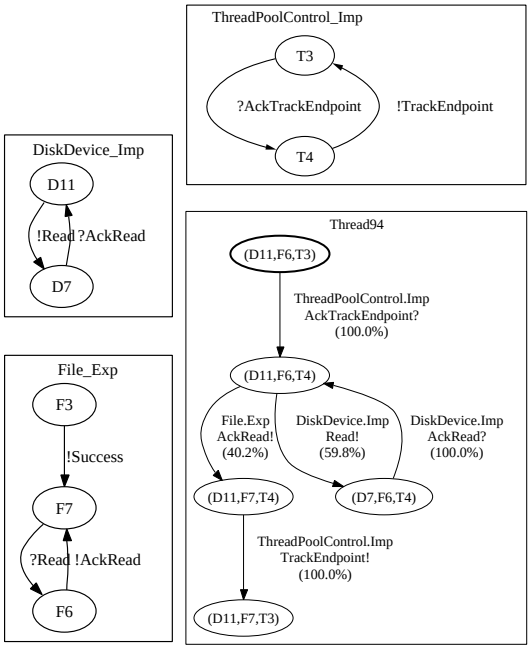

Figure 5: A file system worker thread channel FSMs and the joint PFSM, built from the cross-product of all state tuples and the union of its messages. The messages AckTrackEndpoint and TrackEndpoint can be understood to mean "Do work" and "Finished doing work" respectively. States in the PFSM are labelled with tuples in which each member identifies the current state of each of the three contracts.

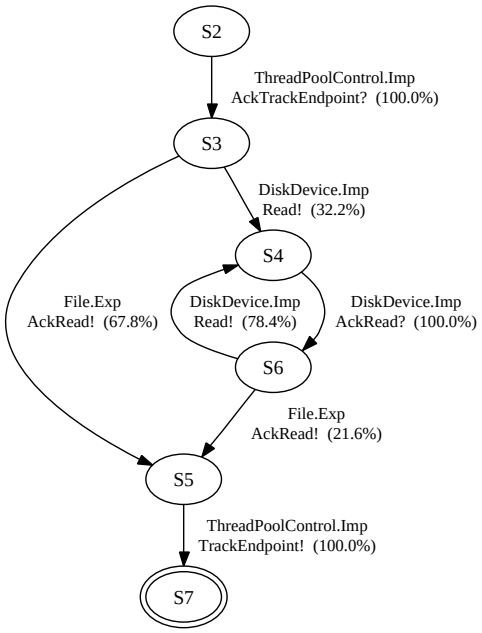

Figure 7: The joint state machine for a WebFiles thread, inferred using the ALERGIA algorithm. Compared to the state machine of Figure 5, ALERGIA has generated an additional state in order to capture the observed probability distribution for disk reads. accesses. The FSM now reveals that two thirds of file requests are served from the cache without going to disk.

that message is in fact received by the thread pool control thread instead.

This simple approach to learning the joint FSM is analogous to program profiling, in that it reveals the proportions in which various transitions occurred for a given execution run. However it doesn't capture the probabilistic distributions of those transitions, which makes it less useful for more sophisticated tasks such as critical path analysis. So, in the Filesystem joint PFSM shown above, we do not have any information about the distribution of the expected number of disk accesses per file request and consequently the proportion of cache hits. In the next section, we explore how grammatical inference can be applied to infer even richer joint PFSMs.

5 Using grammatical inference

In general, this problem can be cast as one of probabilistic *grammatical inference*. Given a set of example strings S, a probabilistic prefix tree acceptor (PPTA) can be constructed that accepts only S. The PPTA is then generalized by merging compatible states. There are many algorithms for doing this in the literature [4], of which ALERGIA[3] is one of the most well-known.

ALERGIA learns a stochastic context-free grammar from a positive set of sample strings in the language by recursively merging states using a statistical compatability measure based on the Hoeffding bound. The resulting grammar gives a compact, if generalized, representation.

In Section 3 we noted that per-contract PFSMs are straightforward to infer by counting the frequency of message types, but that this simple technique loses useful information such as the order in which states occur. In contrast, because ALERGIA operates over "sentences" in the grammar, it is able to discriminate between sequences of states with different statistical properties. Figure 6 contains the Test PFSM produced by ALERGIA. This state machine was derived from the same data that generated the original PFSM of Figure 2, but conveys a great deal more information about runtime behaviour.

Using ALERGIA to generate the joint PFSM further demonstrates that the more sophisticated technique results in richer behavioural models. Figure 7 shows the state machine generated by

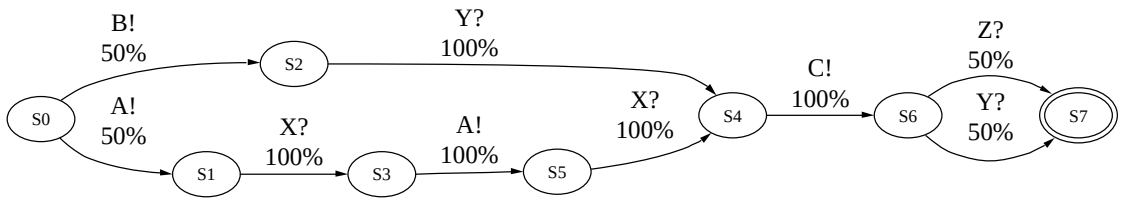

Figure 6: The PFSM inferred for the Test contract by ALERGIA. Compared with the PFSM in Figure 2, the state machine captures much more information about the sequence of messages sent by the test application.

ALERGIA for the WebFiles worker thread that participates in three contract state machines: File, Disk and ThreadPoolControl. In comparison with the model in Figure 5, this PFSM captures the probability distributions of the thread making a request to the Disk Contract in order to fulfil the file `Read` request. We see that 68% of file requests are served from the cache, and the model captures the exponentially decreasing probabilities of making multiple disk reads. Note that from this state machine it is possible to generate strings that could never exist in reality, for example, by recursing infinitely on the DiskRead arcs.

Grammar inference algorithms are designed to find the allowed transitions from a well-defined start state, which is sometimes difficult to identify in a real system. In Figure 6, states `S0`, `S3`, `S4` and `S7` in the SCFG all correspond to the contractual `Start` state, which the thread passes through several times on each iteration. Algorithms such as *Sequitur*[7] efficiently find lexical structure in a single long string by iteratively detecting repeated substrings and replacing them with production rules. The resulting set of rules implicitly identify the start positions of repetitive behaviour and could potentially be used to solve this "start state" problem.

The state machine structure extracted by ALERGIA is very sensitive to the statistical similarity test applied: for instance the loop between `S4` and `S6` in Figure 7 could have been elided using a less refined measure. We hope to address how to set this test appropriately in future work.

6 Conclusion

In this paper we have examined a small number of simple machine learning techniques to try and capture the communication patterns of a complex system at runtime. In Singularity the behaviour of a thread as a whole is bounded by the joint FSM, but runtime monitoring is still necessary to find some of the feasible paths in the state machine and to identify the dominant interactions under the current workload. In a less well-specified system the problem would be much harder, as the underlying state machine would also have to be computed. Related work from the program correctness community on inferring program specifications from the observed runtime behaviour [1, 6] may provide a means of applying this approach to a commodity operating system. Other important future work is to extend the PFSMs with resource demand distributions in order to incorporate performance information in the derived models.

References

[1] G. Ammons, R. Bodk, and J. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 259–272, Dec. 2004.

[3] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence*, pages 139–152. Springer-Verlag, 1994.

[4] P. Dupont, F. Denis, and Y. Esposito. Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*, 38(9):1349–1371, 2005.

[5] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[6] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'05)*, Sept 2005.

[7] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.