Synapse: automatic behaviour inference and implementation comparison for Erlang

Pablo Lamela Seijas, Simon Thompson
Dept. of Computer Science, University of Kent
{p.lamela-seijas,s.j.thompson}@kent.ac.uk

Ramsay Taylor, Kirill Bogdanov, John Derrick
Dept. of Computer Science, University of Sheffield
{r.taylor,k.bogdanov,j.derrick}@dcs.shef.ac.uk

Abstract

In the open environment of the world wide web, it is natural that there will be multiple providers of services, and that these service provisions — both specifications and implementations — will evolve. This multiplicity gives the user of these services a set of questions about how to choose between different providers, as well as how these choices work in an evolving environment.

The challenge, therefore, is to concisely represent to the user the behaviour of a particular implementation, and the differences between this implementation and alternative versions. Inferred models of software behaviour – and automatically derived and graphically presented comparisons between them – serve to support effective decision making in situations where there are competing implementations of requirements. In this paper we use state machine models as the abstract representation of the behaviour of an implementation, and using these we build a tool by which one can visualise in an intuitive manner both the initial implementation and the differences between alternative versions.

In this paper we describe our tool *Synapse* which implements this functionality by means of our grammar inference tool *StateChum* and a model-differencing algorithm. We describe the main functionality of *Synapse*, and demonstrate its usage by comparing different implementations of an example program from the existing literature.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Computer-aided software engineering

General Terms Algorithms, design, documentation

Keywords Grammar inference, comparison, machine extraction, Erlang, Finite State Machines, active learning, passive learning, model differences

1. Introduction

The work described in this paper is part of a set of new mechanisms and tools to support effective decision making in situations where there are competing implementations of requirements. In the open environment of the world wide web, it is natural that there will be multiple providers of services, and that these service provisions — both specifications and implementations — will evolve.

For example, there might be multiple implementations of a particular web service or component, and typically in such a scenario each implementation will partially fulfil the requirement, perhaps in a buggy way. Equally importantly, an evolving system goes through multiple implementations of requirements that are themselves evolving, and it is necessary to revise decisions in the light of these changes. Finally, we aim to address systems which deliver the same core functionality, but which behave differently according to the parametrization or configuration in which they are built.

In this paper we concentrate on the first of these situations, where we encounter differing implementations of a specification, or implementations that fail to meet the specification in one or more ways. We provide here a number of different techniques and tools that attack the problem in complementary ways.

Where there are multiple implementations available, software engineers need to be given a suitably high-level and intuitive understanding of the behaviour of each implementation, and of the differences between implementations. Previous work has demonstrated the practicality of automatically inferring state machine models of software behaviour [9, 10, 13, 15, 16]. These models offer easy to comprehend representations of a system's behaviour, and offer a sufficiently abstract platform on which to discuss the differences between implementations in a systematic but intuitive way [6]. The work presented in this paper builds on top of the existing techniques for extracting models from systems, as implemented in *StateChum*, including the *PTSLDiff* algorithm and the Erlang module inference system [14].

Section 2 introduces the *Frequency Server* example [8] used in our earlier work [2, 4], which will be used as our running example. In Section 3 we provide some background on grammar inference and difference deduction and their application to Erlang.

In Section 4 we describe our tool that integrates these techniques into an Erlang framework called *Synapse*, that provides an Erlang interface to these facilities. It allows models to be inferred either passively from traces of an implementation, or actively from an Erlang module, and the resulting model is presented as an Erlang data structure. This can then be compared to another model and the differences recorded and visualised. By presenting both an Erlang interface to the learning and differencing mechanisms together with an Erlang representation of the results, the results of this work can be integrated into existing Erlang development workflows, and used by existing Erlang analysis tools.

As a guide to the use of Synapse, in Section 5, we present a set of variants of the *Frequency Server* example. We see from this that the differences between two implementations can be visualised as the "diff" between two machines, highlighting the set of changes that needs to be made to go from one machine to the other. Finally, in Section 6, we conclude.

2. Running example

Our running example is a simple program extracted from *Erlang Programming* [8], called `frequency_server`. The program represents an automatic service for spectrum management that allocates frequencies on demand on a first come, first served basis. A finite set of frequencies is offered and the server guarantees that each frequency is allocated to at most one client. Whenever a client is no longer interested in a frequency that it has previously allocated, it may communicate this to the server using the command `deallocate`, this way deallocated frequencies may be reused by other clients.

2.1 Base implementation

In Listing 1, we provide the source code for the system that we use as base implementation in the rest of this section. To avoid redundancy, the code for the other configurations will be expressed as modifications to this base implementation.

```
%% Code from
%% Erlang Programming
%% Francecso Cesarini and Simon Thompson
%% O'Reilly, 2008
%% http://oreilly.com/catalog/9780596518189/
%% http://www.erlangprogramming.org/
%% (c) Francesco Cesarini and Simon Thompson
%% Modified by: Pablo Lamela on Dec 2013

-module(frequency).
-export([start/0, stop/0, allocate/0,
         deallocate/1]).
-export([init/0]).

%% These are the start functions used to
%% create and initialize the server.

start() ->
    register(frequency, spawn(frequency,
                              init, [])).

init() ->
    process_flag(trap_exit, true),
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11].

%%  The client Functions

stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate,Freq}).

%% We hide all message passing and the
%% message protocol in a functional
%% interface.
call(Message) ->
    frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

reply(Pid, Message) ->
    Pid ! {reply, Message}.

loop(Frequencies) ->
    receive
        {request, Pid, allocate} ->
            {NewFrequencies, Reply} =
              allocate(
                sortfreqs(Frequencies),
                Pid),
            reply(Pid, Reply),
            loop(NewFrequencies);
        {request, Pid, {deallocate,Freq}} ->
            NewFrequencies =
              deallocate(Frequencies, Freq),
            reply(Pid, ok),
            loop(NewFrequencies);
        {'EXIT', Pid, _Reason} ->
            NewFrequencies =
              exited(Frequencies, Pid),
            loop(NewFrequencies);
        {request, Pid, stop} ->
            reply(Pid, ok)
    end.

sortfreqs({Freqs, Allocated}) ->
    {lists:sort(Freqs), Allocated}.

allocate({[], Allocated}, _Pid) ->
    {{[], Allocated},
     {error, no_frequencies}};
allocate({[Freq|Frequencies], Allocated},
         Pid) ->
    link(Pid),
    {{Frequencies,[{Freq,Pid}|Allocated]},
     {ok,Freq}}.

deallocate({Free, Allocated}, Freq) ->
    {value,{Freq,Pid}} =
      lists:keysearch(Freq,1,Allocated),
    unlink(Pid),
    NewAllocated=lists:keydelete(Freq,1,
                                 Allocated),
    {[Freq|Free],  NewAllocated}.

exited({Free, Allocated}, Pid) ->
    case lists:keysearch(Pid,2,Allocated) of
        {value,{Freq,Pid}} ->
            NewAllocated =
              lists:keydelete(Freq,1,
                              Allocated),
            {[Freq|Free],NewAllocated};
        false ->
            {Free,Allocated}
    end.
```

Listing 1: *Listing for base implementation*

2.2 Alternative implementations

With this base implementation in mind, we can think of several variants of this, including the following.

1. Regarding the implementation of `deallocate`: what should happen whenever a client tries to deallocate a frequency it has not allocated?

(a) `noop`: the server must act as if deallocation was successful but do nothing.

(b) `cannot`: the server must crash by throwing an exception and returning an error.

2. Regarding the order of frequency allocation: in what order must frequencies be allocated?

   (a) `smallf`: the server must always allocate the smallest available frequency first.

   (b) `lifo`: the server must always allocate first the most recently deallocated frequency. If the frequencies have never been allocated yet, allocation starts with the highest frequencies.

3. Regarding the initial number of frequencies: before starting the `frequency_server` we must make a number of frequencies available, which for our purposes we consider finite. In this paper we consider two possibilities:

   (a) 2 initial frequencies: which we will represent with numerals `10` and `11`.

   (b) 3 initial frequencies: which we will represent with numerals `10`, `11`, and `12`.

From now on we will represent any set of configurations as an Erlang tuple in the form `{DeallocationMethod, Allocation Method, InitialFrequencies}`. Since it is the simplest, we will use `{cannot, smallf, 2}` as the base configuration for comparisons.

For example, we might be given the code for the deallocation behaviour. This is given as a modification of the implementation above, so that it does not produce an error when a client tries to deallocate a frequency that is not allocated, see Listing 2. The differences here are highlighted by red / green representing removal / addition respectively - of course the challenge in practice is that usually one doesn't know precisely which bits of code have been altered.

```
deallocate({Free, Allocated}, Freq) ->
    {value,{Freq,Pid}} = lists:keysearch(
                            Freq,1,Allocated),
    unlink(Pid),
    NewAllocated =
      lists:keydelete(Freq,1,Allocated),
    {[Freq|Free], NewAllocated}.
    case lists:keysearch(Freq,1,Allocated) of
      {value,{Freq,Pid}} ->
          unlink(Pid),
          NewAllocated =
            lists:keydelete(Freq,1,
                            Allocated),
          {[Freq|Free], NewAllocated};
      _ -> {Free, Allocated}
    end.
```

Listing 2: *Modifications for* `noop` *deallocation*

What we wish to do now is to both calculate the difference between this new implementation and the original one, and then represent visually these alterations to the user in the most intuitive manner possible. To do this we will use grammar inference techniques to build models of the original system, the variant, and the differences between these.

3. Background work: grammar and difference inference

The previous section has introduced the example Erlang implementation of the frequency server, and some potential variants. This section will introduce the techniques by which an implementation's behaviour can be automatically inferred, producing a Finite State Machine model of the behaviour that is easily understood by a software engineer who wants to understand the system's operation.

Our previous work has demonstrated the use of grammar inference techniques to infer models of behaviour for Erlang implementations [7, 14]. Grammar inference techniques work by considering sequences of operations in the system — hereafter referred to as *traces* of the system. A sufficiently diverse set of such traces (perhaps derived from log data or from unit tests) can be considered the *language* of the system, and techniques exist to infer a *grammar* for this language in the form of a state machine that accepts the valid traces and rejects the invalid traces. This state machine is then an effective abstract model of the system's behaviour [10].

3.1 Grammar Inference

Algorithmic approaches to language learning started in the 1960s with Gold [11], building on earlier work to formalise natural language. In the 1980s, Angluin published the L*-algorithm [1] that learns a language from an expert oracle by presenting queries. The queries take the form of sequences that may or may not be in the language, and the expert oracle replies simply whether they are or are not valid language elements. The L*-algorithm queries the language incrementally and exhaustively, attempting each alphabet element from each state and iterating until no new states are identified. This produces the complete automaton but requires lengthy exploration of the language and heavy use of an expert oracle.

Later work has aimed to learn state machine language representations from partial data – usually a subset of the possible sequences of the language – without user queries. Evidence driven state merging (EDSM) algorithms such as BlueFringe [12] operate on a set of positive and negative traces from a system or language and produce a state machine that accepts positive traces and rejects (in the form of a transition to a failure state) negative traces.

The inference algorithm starts by constructing what is called a Prefix Tree Acceptor (PTA) which is a tree that represents exactly the traces given to a learner. Taking the first three positive traces above, the corresponding PTA is shown at the top of Figure 1. Here path `start,allocate/1,deallocate(1)` corresponds to a trace `+ start allocate/1 deallocate(1)`. Trace `+ start allocate/2 deallocate(2)` is assumed to have entered the state after `start` and added two more states. Note that the second positive trace is subsumed by the third one.

The problem of inference from such a graph is identification of loops; where one is confident that two states correspond to the same state of a target graph, they can be merged into a single state. In the figure, there are two states from which the path `allocate/1, deallocate(1)` can be followed, suggesting a commonality in their behaviour. It does not necessarily mean that these states correspond to the same target state but is often a good indication of this. The outcome of merging them is shown at the bottom of the figure. A well-known method of model inference called K-Tails [5] is based on this observation and merges pairs of states where all outgoing paths of length k are the same. The value k is the parameter that determines how much is merged; with a value of zero, one ends

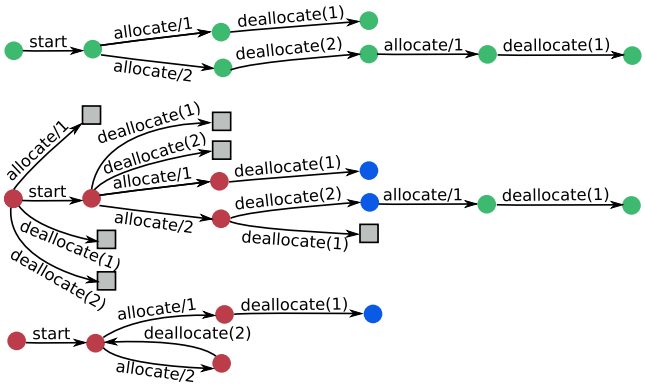

Figure 1: *Illustration of the model inference*

up with a single-state automaton, with a very large k, no mergers are possible and the PTA is returned.

The passive inference method used in this work to infer models (EDSM from [13]) is more complex: it is using an idea of a *blue fringe* to control the exploration of PTA and an idea of *scores* to rank potential mergers. In addition, reject-paths are used to block clearly erroneous mergers. The idea of scores is to evaluate the number of states that will be removed if a pair is merged. In the above example, the merger turned a PTA into a graph with two fewer states and hence the score will be 2. Where a pair of states have a lot in common, their merger is more likely to be correct compared to a pair with little in common; a merge of such a pair is also likely to significantly reduce the graph size. This is why scoring is based on the reduction of graph size.

It is easy to encounter many states with little in common. This is particularly true for the tail-end states of a graph where scoring would give a zero which makes it hard to tell whether to merge or not. The presence of paths leading to a reject-state is key to effective learning. The middle graph of Figure 1 contains a few negative traces from the above-mentioned set of traces added to the top PTA. Reject-states are drawn as rectangles. A merger is not permitted to merge a normal (positive) state and a reject-state. For instance, where one attempts to merge the source and target states of the `start` transition, this implies that `allocate/1` should be both possible (it is present from the target state of the `start` transition) and impossible (it is present as a reject-transition from the initial state). In such cases, no merger is performed and the score is assumed to be negative.

Merging is improved where one uses a systematic approach to exploration of a graph with the aim to consider pairs of states that have higher scores first. The idea of a blue fringe [13] is to view the graph as a collection of states that were considered and are not mergeable pairwise (*red states*) and then evaluate a boundary of this set of states (called *blue fringe*) as candidates for mergers with some of the red ones. Where a blue state cannot be merged (which is the case when the score between it and all of the red states is negative), it is coloured red and the new blue boundary is computed. Initially, the set of red states contains only the initial state.

In the running example, the initial state is red and the blue fringe only contains the target state of `start`. Such a blue state cannot be merged into the initial state hence it is coloured red and the two target states of `allocate`-transitions are made blue. These are once again have to be coloured red and the outcome is shown in the middle of the figure. The target state of `deallocate(2)` can be merged into the target state of `start` with a score of 2; the remaining blue state can be merged into any red one with each such merger scoring a zero. A pair with a higher score is chosen and the inference arrives at the graph at the bottom of Figure 1. Ideally, one would proceed to (erroneously) merge the remaining blue state with the initial state. This can be most easily handled either by using a larger range of positive traces or by using a threshold where only pairs with scores above it are merged. In this work, the former approach is used.

The BlueFringe algorithm [13] was shown to be highly effective in the *Abadingo One* competition for learning algorithms, and does not require the use of an expert oracle, operating exclusively on the positive and negative traces presented. The QSM algorithm [9] reintroduces the oracle query approach by producing queries from the new traces that are possible in the system after the proposed merge and requires the oracle to either confirm that they are positive traces, or supply the shortest negative prefix. If the response is the same as the classification of those traces by the outcome of the proposed merge then QSM will continue to suggest merges. If the response contradicts the merged automaton the algorithm restarts from the beginning, with this additional negative trace. Eventually enough information is accumulated for the learning process to converge.

QSM does not aim to learn the complete automaton, its purpose is to generalise the supplied traces. It can be supplied a subset of traces with "interesting" behaviours, and it will generalise from these using queries. Consequently, it is more tolerant of initial trace sets that are not representative samples of the system's behaviour, so long as they contain enough significant features to allow query generation to operate.

Non-exhaustive language learning algorithms have significant application to software testing as they present an opportunity to explore the behaviour of the software system, without the resource and time requirements of exhaustive model checking. Additionally, passive inference can be performed on existing program traces — perhaps from system log data, or by instrumenting existing tests — and active inference techniques can derive suitable parameter values from this existing information. As well as providing a representation of the behaviour of the system, this can provide a measure of the test or trace set used, since a test set that explores more of the potential behaviour of a system will produce a more detailed and/or accurate model during inference. Using behaviour inference for test assessment was first suggested by [19], and has been shown to provide a better metric of *test adequacy* [10].

3.2 The tools *StateChum* and *PTSLDiff*

The *StateChum* [7] system was developed to implement the QSM algorithm with the objective of reverse engineering state machine representations of software behaviour from software trace data [16]. *StateChum* takes sets of positive and negative traces and treats them as a *prefix closed* language — so, if any trace is accepted by the system then all prefixes of that trace must also be accepted; for any negative trace of the system all prefixes are positive traces but the final event "crashes" the system and cannot be recovered. The positive traces are composed into a *prefix tree automaton* that starts from an initial state and adds branches as traces diverge.

StateChum also includes the implementation of a model differencing tool *PTSLDiff* produced at the University of Sheffield [6]. This allows two FSM models to be compared and computes a *structural difference*. The difference is expressed in terms of the smallest set of modifications required to convert the first state machine into the

second. Modifications include adding and removing states, adding and removing edges, and renaming of states and edges.

Most traditional methods of FSM comparison expect the initial states of the two to match; in software models, there is nothing preventing extended initialisation routines to be added to a new version which means that differencing cannot have an a-priori knowledge where to start. The idea of differencing is (1) to compute pairwise scores between states of the graphs to compare, followed by (2) identification of landmarks and subsequent (3) construction of a difference. Where a pair of states has most of the outgoing transitions in common, this is deemed a better match than the one with only a fraction of outgoing transitions that match; moreover, where matching transitions lead to states considered similar, this is a better case compared to target states being very different. This is why a score between a pair of states depends on scores of pairs of states reached by matched transitions; this can be seen as a system of linear equations and solving it gives a good guess of state similarity. It can also be computed both in forward and inverse direction and results averaged. Landmarks are pairs which are not only well-matched but also where any of the two states in a pair are not well matched to any other state (an idea also known as 'stable marriage'). Construction of a difference starts by choosing such well-matched pairs and then propagating matches along matched transitions that lead to pairs with best scores; this is done both in forward and inverse direction. This corresponds to a traveller matching the landscape around them to a map and looking at streets in a similar direction, expecting them to lead to matched places. For each new matched pair, matched outgoing transitions are considered and so on. After such an partial injective map between states of two considered graphs is computed, all transitions that do not match are to be either added or removed and are hence are included in the difference. The construction of a map is aimed to construct a good match so as to minimise the number of transitions to add or remove; it is not guaranteed to always find the smallest difference. It is guaranteed, however, to construct a valid difference, in that it can be applied to the first graph and this would yield a graph isomorphic to the second one. The differencing works on arbitrary directed graphs, however in this work is it only applied to deterministic ones.

Finally, *StateChum* contains code to visualise models and model differences. All of these features are accessible through the *Synapse* interface that is the focus of the next section.

4. The *Synapse* interface

StateChum provides various powerful features for model inference, model differencing, and visualisation. To leverage this toolset for Erlang applications, and to allow its integration into automated Erlang development processes, this work has developed the *Synapse* tool. *Synapse* provides a simple and modular Erlang interface to grammar inference and Finite State Machine analysis tools. Currently, *Synapse* interfaces to the *StateChum* tool allow for passive and active inference, and for a range of visualisation and differencing functions. The framework has been designed in a modular fashion to allow easy integration of other learning or analysis tools in the future.

The *Synapse* interface provides separate functions for each stage of a typical behaviour inference workflow, allowing time-consuming steps such as model inference to be performed once, and the results to be stored or manipulated as Erlang terms. The principal operations supported are: *learning* (both active and passive), *FSM visualisation*, *model differencing*, and *difference visualisation*.

4.1 Model inference from traces

Synapse provides a flexible interface to the *StateChum* inference system. There are parsing functions to support several trace formats, including the format used by the STAMINA competition [17], and the parametrised format used in Mint [18].

These traces must demonstrate sequences of operations in the implementation that is being studied. The level of abstraction chosen will determine the level of abstraction of the inferred model. These traces could be sourced by extracting sessions from log data, by instrumenting unit tests, or by random testing with a tool such as QuickCheck [3].

The traces should be classified as *positive* and *negative* depending on whether the model should accept them or reject them. *StateChum* operates on *prefix closed* languages, so any negative traces should be positive traces up until the last operation. Defining *negative* is an abstraction decision: it could represent operations that will crash the system, cause an exception, or operations that are simply not available at that point.

Currently, *StateChum* builds Finite State Machine models, that do not have data or parameter components. Although there is work being undertaken to infer *Extended Finite State Machine* models of software (such as [18]) that is not considered here. The example operations used in this paper use a limited set of parameters and make these explicit in the operations — so `deallocate(1)` and `deallocate(2)` are considered to be entirely distinct operations.

Some STAMINA format traces from the base implementation described in Section 2 are shown below. These represent function calls, with any arguments used in brackets (`()`), and the result of the function call after the slash (`/`). Where no return value is specified the model ignores the return value, and would accept any run of the system that performs the operation with the specified parameters. Where a return value is specified, this is the return value expected at this point. Non-deterministic systems cannot be modelled by *StateChum*.

```
+ start allocate/1 deallocate(1)
+ start allocate/2 deallocate(2)
- start allocate/2 deallocate(1)
- allocate/1
- deallocate(1)
- allocate/2
- deallocate(2)
- start deallocate(1)
- start deallocate(2)
- start allocate/2 deallocate(2) deallocate(1)
+ start allocate/2 deallocate(2) allocate/1
deallocate(1)
+ start allocate/1 deallocate(1) allocate/2
deallocate(2)
+ start allocate/2 allocate/1 deallocate(2)
deallocate(1)
+ start allocate/1 allocate/2 deallocate(1)
deallocate(2)
```

These can be presented to *Synapse* and are parsed into Erlang terms as:

```
Traces =
[{pos,[start,'allocate/1','deallocate(1)']},
 {pos,[start,'allocate/2','deallocate(2)']},
 {neg,[start,'allocate/2','deallocate(1)']},
 {neg,['allocate/1']},
 {neg,['deallocate(1)']},
```

```
 {neg,['allocate/2']},
 {neg,['deallocate(2)']},
 {neg,[start,'deallocate(1)']},
 {neg,[start,'deallocate(2)']},
 {neg,[start,'allocate/2','deallocate(2)',
       'deallocate(1)']},
 {pos,[start,'allocate/2','deallocate(2)',
       'allocate/1','deallocate(1)']},
 {pos,[start,'allocate/1','deallocate(1)',
       'allocate/2','deallocate(2)']},
 {pos,[start,'allocate/2','allocate/1',
       'deallocate(2)','deallocate(1)']},
 {pos,[start,'allocate/1','allocate/2',
       'deallocate(1)','deallocate(2)']}]
```

Similar traces could be produced directly by an Erlang testing tool. These can be stored or manipulated as Erlang terms before being presented to the inference component to produce an FSM.

To infer a state machine model of the system these traces are passed to *StateChum* through the *Synapse* interface:

```
SM1 = synapse:passive_learn(Traces).
```

This passes the trace set above to *StateChum* with the default configuration options necessary for passive learning. The function `synapse:learn` can take additional parameters that control the behaviour of the learner for specialist tasks.

Additionally, *Synapse* provides an interface to the *StateChum* Erlang active inference mechanisms through the `learn_erlang` function. This can be called with a path to an Erlang source file to apply the Erlang module inference described in [14].

The return value of all of the `learn` functions is an Erlang record representing the state machine that was inferred.

```
-record(statemachine,{
  states :: list(state()),
  transitions :: list(transition()),
  initial_state :: state(),
  alphabet :: list(event())
 }).
```

For the small set of traces above this is a simple machine:

```
SM1 = synapse:passive_learn(Traces).
Progress: 24 states
Progress: 20 states
Progress: 17 states
Progress: 15 states
Progress: 14 states, 14 red states
Progress: 12 states
Progress: 11 states
Progress: 10 states
Progress: 9 states
Progress: 8 states, 8 red states
Progress: 7 states
Progress: 6 states
{statemachine,['P1000','N1000','P1001','P1004',
               'P1002','P1008'],
              [{'P1000','allocate/1','N1000'},
               {'P1000','allocate/2','N1000'},
               {'P1000','deallocate(1)','N1000'},
               {'P1000','deallocate(2)','N1000'},
               {'P1000',start,'P1001'},
               {'P1001','allocate/1','P1004'},
               {'P1001','allocate/2','P1002'},
               {'P1001','deallocate(1)','N1000'},
               {'P1001','deallocate(2)','N1000'},
               {'P1004','allocate/2','P1008'},
               {'P1004','deallocate(1)','P1001'},
               {'P1002','allocate/1','P1008'},
               {'P1002','deallocate(1)','N1000'},
               {'P1002','deallocate(2)','P1001'},
               {'P1008','deallocate(1)','P1002'},
               {'P1008','deallocate(2)','P1008'}],
              'P1000',
              ['allocate/2',start,'deallocate(1)',
               'deallocate(2)','allocate/1']}
```

This encodes the states, initial state and transition matrix, as well as the alphabet, into a form that can be stored or manipulated by other Erlang based tools. The *negative* state is always encoded as `N1000`. This state is the endpoint for the last event in any negative traces.

As well as *passive learning* — building a state machine from supplied traces — *Synapse* provides an interface to *StateChum's active learning* systems [14]. This approach can use the exported interface of an Erlang module, or the standard *call* and *cast* operations of an OTP behaviour and apply the QSM active learning algorithm to infer the module's behaviour. The system will load the selected module and determine the available API functions. These are combined in random orderings to produce some initial traces for the learner, but as the QSM algorithm progresses it produces queries, which can be answered by executing the traces as tests on the module itself. The active learning process can produce a more accurate state machine, as it is able to improve its state merging by checking potential merges through interactive queries answered by a user. However, it requires that the Erlang module can be executed, and is relatively independent of other modules and behaves deterministically.

4.2 Visualisation of the inferred model

Synapse allows Erlang programmers to utilise the visualisation components of *StateChum* independently of the learning process. This allows time consuming inference processes to be run once and the results stored, manipulated, and displayed independently.

The *StateChum* visualisation is initiated by calling the function `synapse:visualise`. The visual interface allows the user to rearrange the state machine, and save and load layouts. Several visualisation windows can be opened simultaneously, allowing visual comparison of state machines. This is further aided by the layout saving mechanism, which allows a layout to be applied to different machines with the same state names. This places the states in the same configuration on two different FSMs, making visual comparison easier.

By default, the negative state and the final ("crashing") transitions leading to it are hidden to make the diagram more readable, but this can be changed either by adding parameters to the `visualise` function, or through the user interface. This is shown in Figure 2, depicting the visualisation of the model inferred from the above set of traces, including a negative state visible in the top right corner.

4.3 Measuring and visualising model differences

Model inference techniques also allow us to record and then visualise the differences between implementations. Specifically, the structural differencing algorithms described in [6] are implemented in *StateChum*, and the *Synapse* interface allows these to be used on stored Erlang LTS representations.

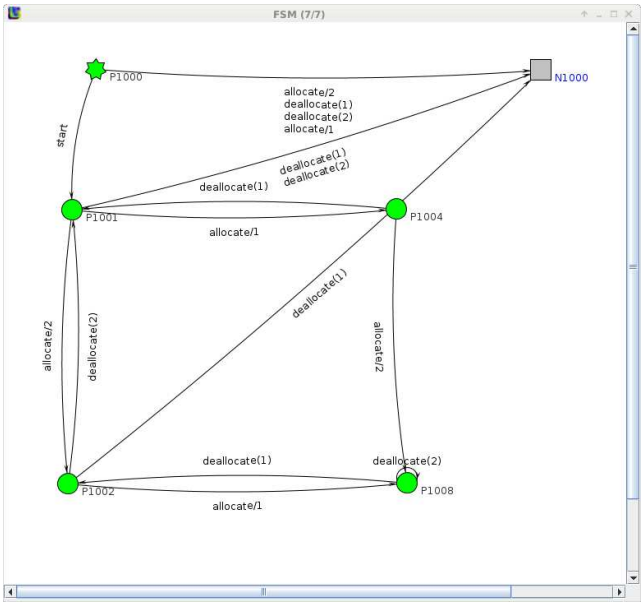

Figure 2: *FSM visualisation with negative state*

The `synapse:diff` function takes two state machine record representations and returns an Erlang record representing the structural difference computed as described in Section 3.2.

```
-record(statemachinedifference,{
  added_transitions :: list(transition()),
  deleted_transitions :: list(transition()),
  added_states :: list(state()),
  deleted_states :: list(state()),
  name_mapping :: list({state(),state()}),
  initial_state :: state()
 }).
```

For example, adding an extra frequency to the very short set of traces above and inferring a new machine (`SM2`):

```
Diff = synapse:diff(SM1,SM2,[]).
Synapse started.
StateChum at <0.62.0>
{statemachinedifference,
    [{'P1000','allocate/3','N1000'},
     {'P1000','deallocate(3)','N1000'},
     {'P1001','allocate/3','P1008'},
     {'P1001',allocate2,'P1000'},
     {'P1008','deallocate(3)','P1008'}],
    [],[],[],
    [{'P1004','P1005'},{'P1008','P1015'}],
    'P1000'}
```

This model difference can be stored and manipulated as an Erlang term. This allows meta-differencing, or difference metrics to be calculated and fed back to testing tools.

Having produced an Erlang representation, the final component of *StateChum* that is available through the Synapse interface is difference visualisation. This allows the difference between two FSM models to be visualised in a single image. The function `synapse:visualise_diff` requires an FSM record for the "original" state machine and a difference record representing the changes to move from the "old" to the "new" state machine.

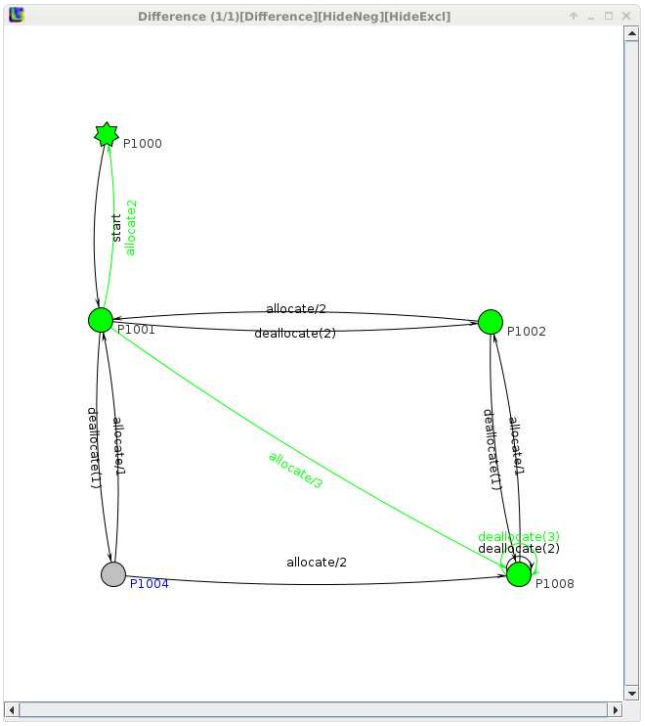

Figure 3: *Difference visualisation*

Added states and transitions are labelled green, removed states and transitions are labelled red. This allows a user to quickly identify areas of difference, and it can provide some insights into the effects of particular changes that may not be apparent from a review of the source code. For example, Figure 3 visualises the differences between the base implementation and the implementation with three frequencies: it is thus the visualisation of the difference record above.

5. Applying *Synapse* to the Frequency Server

This section illustrates the use of *Synapse* as applied to our running example. As described in Section 4, *Synapse* provides an Erlang interface to the grammar inference and model differencing implementations in *StateChum* and *PTSLDiff*. We discuss three variants of the base implementation described in Section 2. For each of these we will show three models representing the original implementation, the variant and the difference. The first two are generated by using *StateChum*'s active learning from two independent configurations of the `frequency_server`; the third diagram is generated by using *PTSLDiff* on the first two, and shows which changes have to be done in the first model to transform it into the second.

In the model generated by PTSLDiff, which from now on we will call the *diff diagram*, added transitions are represented in green, removed transitions are represented in red, and transitions that are common to both diagrams are represented in black. Nevertheless, all the states are represented in green, we already know that states that only have green transitions are added states, and that states that only have red transitions are deleted states. The initial state — the state in which the `frequency_server` is before it is started — is represented by a 7-pointed star. Illegal transitions — transitions that would produce an exception — are omitted from the diagram for clarity.

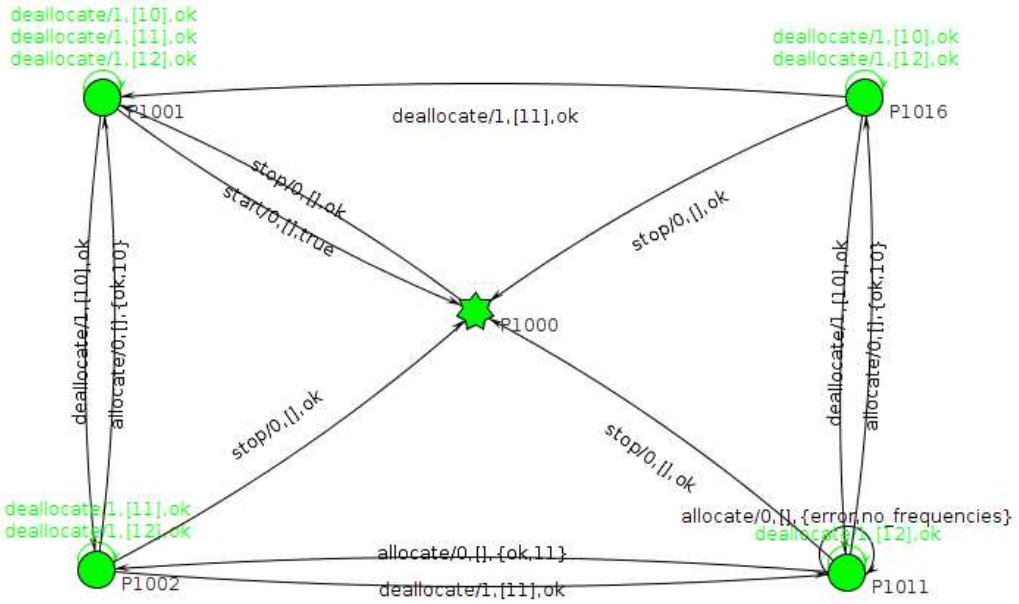

Figure 6: *Differences between* `cannot` *and* `noop` *deallocation*

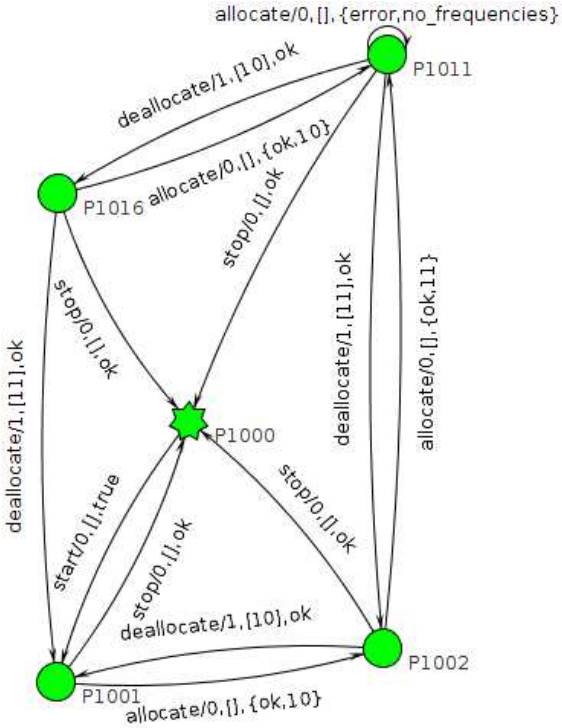

Figure 7: *Base* `smallf` *configuration*

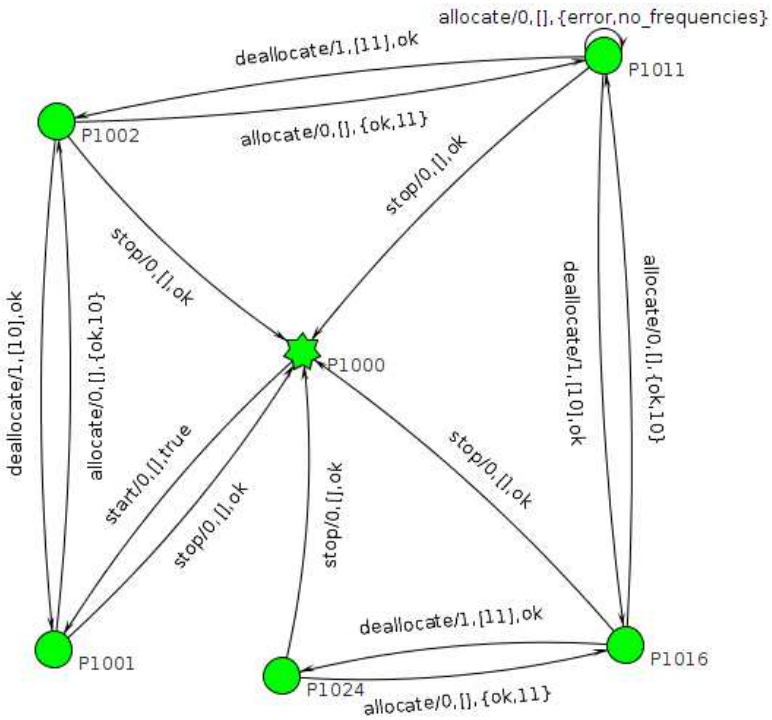

Figure 8: *Alternative* `lifo` *deallocation*

5.1 Deallocation behaviour

The first differences that we consider are in deallocation behaviour. To illustrate this configuration we use the modification given above in Listing 2 (in Section 2), where the code does not produce an error when a client tries to deallocate a frequency that is not allocated.

In Figure 4 we show the model for the base configuration `{cannot, smallf, 2}` and in Figure 5 we show the behaviour for the alternative deallocation implementation `{noop, smallf, 2}`. We can see that the models are very similar in this case. In the middle we have the initial state, and every call to `stop` ends up in it. The other

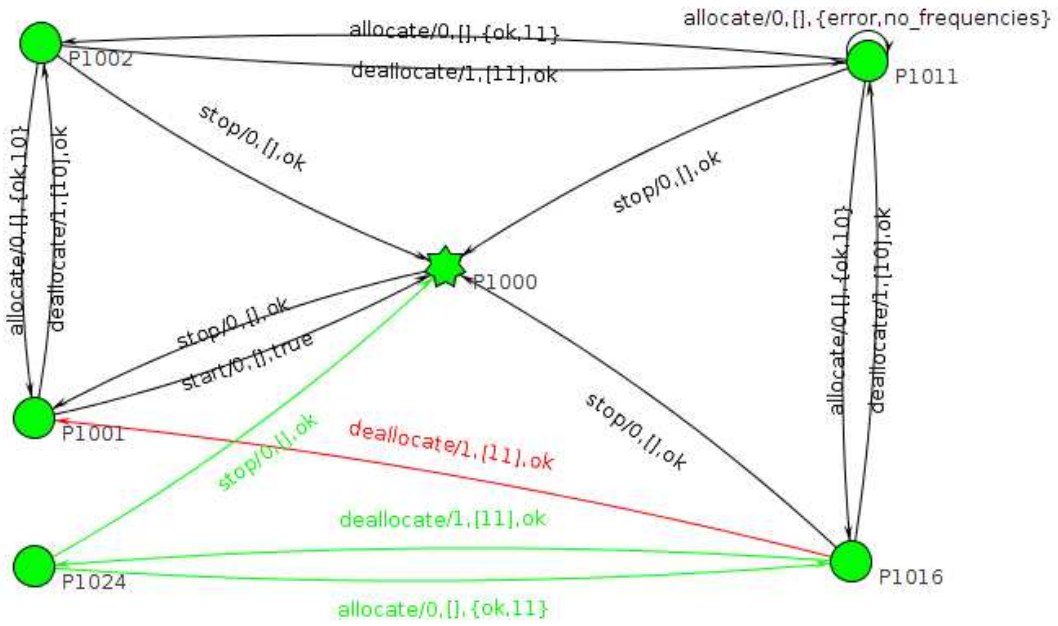

Figure 9: *Differences between* `smallf` *and* `lifo` *allocation*

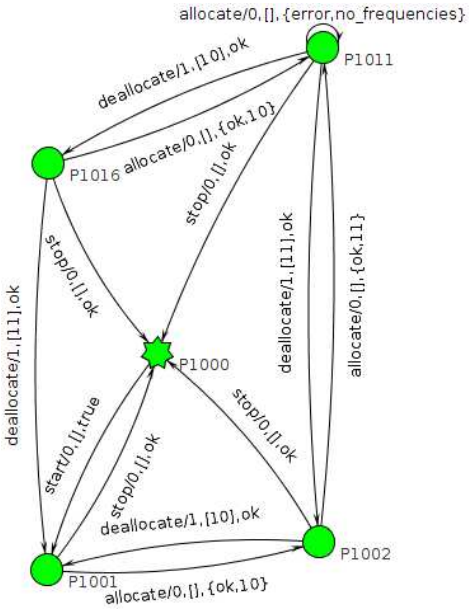

Figure 10: *Base 3-freq conf.*

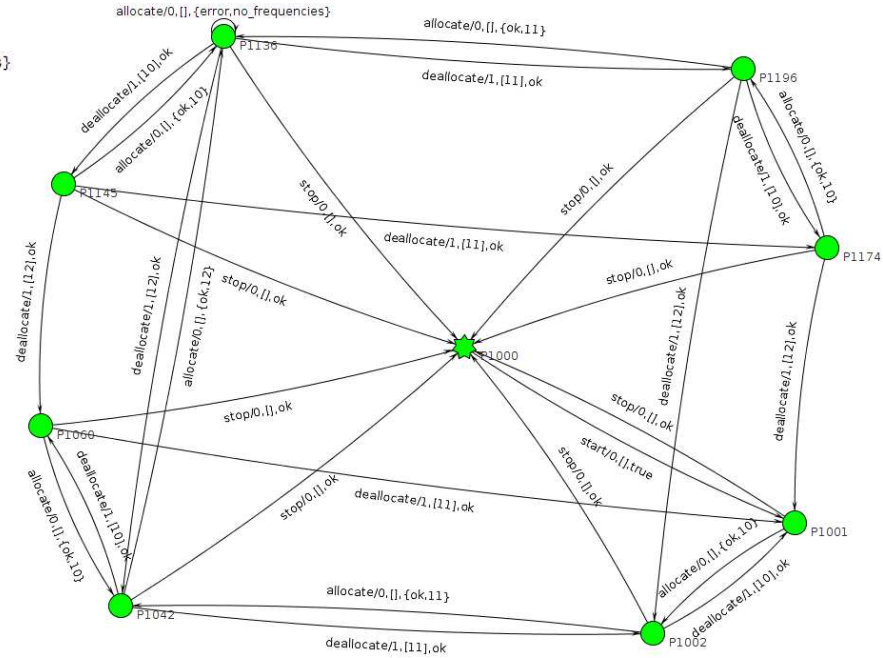

Figure 11: *Alternative configuration with 4 frequencies*

four states represent all the possible sets of available frequencies that can occur with two existing frequencies: `10` and `11`.

We can identify the state with no allocated frequencies in the end of the `start` transition, and we can identify the state with no free frequencies by finding the self-transition of `allocate` that returns `{error, no_frequencies}`.

We can easily see in the *diff diagram*, (Figure 6), that the only effect of the configuration `noop` is the self-transitions that try to deallocate frequencies that are already deallocated, since in the `cannot` configuration these produce an exception and, thus, they are omitted from the diagram.

These three diagrams succinctly represent the behaviour of the two implementations, and the difference between them. The colour

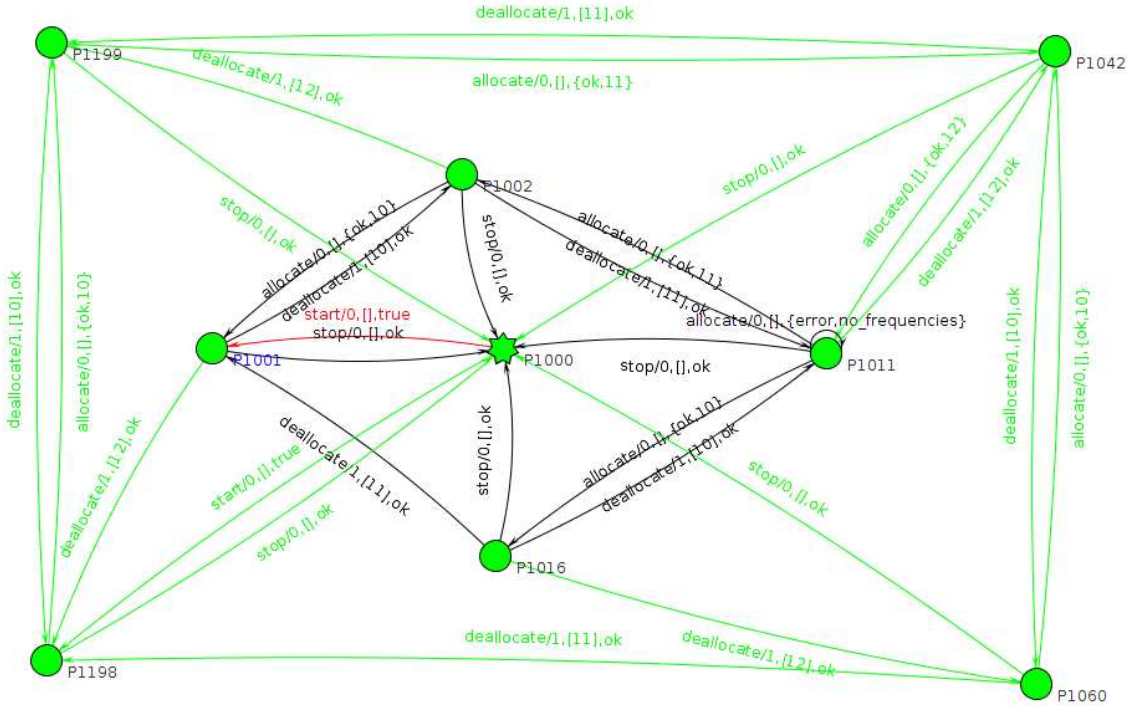

Figure 12: *Differences between using 2 and 3 frequencies*

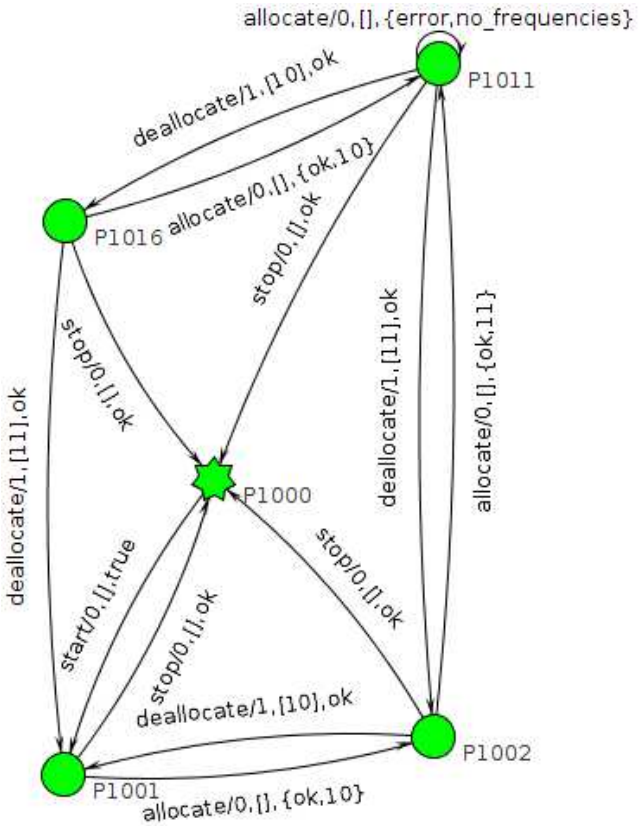

Figure 4: *Base* `cannot` *configuration*

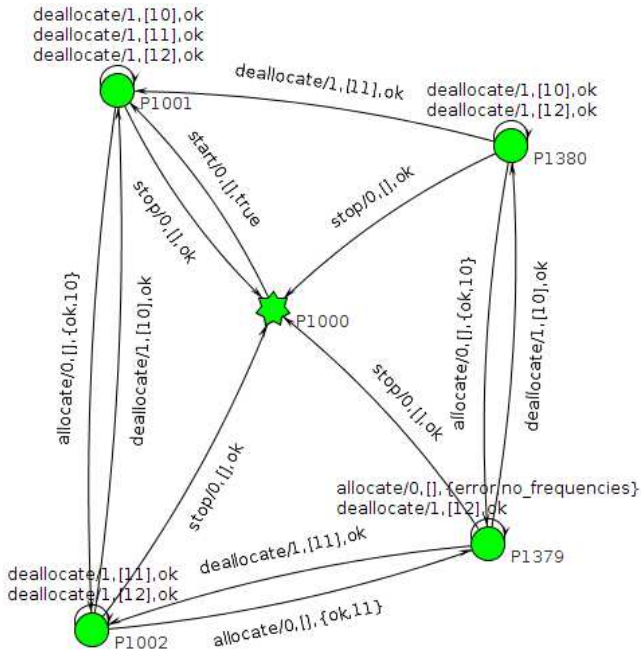

Figure 5: *Alternative* `noop` *deallocation*

coded *diff diagram* is ideal for focussing the attention of software engineers on only those parts of the system that are altered. This approach does so at the *behaviour* level, rather than a simple code diff, which could highlight several disparate areas of the source that require some considerable understanding to link. This simplicity and informative power is the critical contribution of this work.

5.2 Allocation behaviour

Now we illustrate the effect of different allocation implementations. In order to do that, we use again the configuration `{cannot, smallf, 2}`, this time as an example of `smallf` configuration, which we show again in Figure 7 for clarity, and we use `{cannot, lifo, 2}`, (shown in Figure 8), as an example of `lifo` configuration, the modifications made on the base configuration are listed in Listing 3 (illustrating code differences by red and green arcs).

We can intuitively guess that the `lifo` configuration needs a higher number of states because the order in which frequencies were deallocated must be stored too. In Figure 8 we can indeed see that an extra state `P1024` exists without any allocated frequencies. But unlike state `P1001`, calling `allocate` from this new state will allocate `11`, instead of `10`.

In Figure 9 we can clearly see that deallocating `11` after `10` produces a difference between the `lifo` and `smallf` configurations.

5.3 Number of initial frequencies

Finally, we illustrate the effects of increasing the number of initial frequencies. Once more we use the configuration `{cannot, smallf, 2}` as base (Figure 10), and this time we use the configuration `{cannot, smallf, 3}` as target (Figure 11). This is the result of a small modification in the base code, see Listing 4.

We can guess that even though the order of deallocations does not matter because we are using a `smallf` allocation configuration, we will have considerably more states because of the new added frequency. In particular, from the state in the diagram we must be able to tell unequivocally whether we may or may not have allocated each of the possible frequencies, giving $2^3 = 8$ states plus the initial state. Indeed, the diagram for three frequencies is a bit more complicated than usual, but we can still manually identify some similarities, such as the self-transition returning `{error, no_frequencies}`.

This time the use of *Synapse* produces a *diff diagram* (Figure 12), that shows much more information. In addition, the colour code helped us in finding a more intuitive layout.

With this new layout we can immediately see that the behaviour with two frequencies is a subset of the behaviour with three frequencies. In the moment we allocate the frequency `12`, the system behaves as only two frequencies existed, as long as we do not deallocate the frequency `12` or re-start the server, which is of course a coherent behaviour.

But if we look closer, we can also see that the behaviour in green is also analogous to the behaviour in black, with the difference that we can transition to the inner black section if we call `allocate` when both frequencies `10` and `11` are already allocated.

```
loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
      {NewFrequencies, Reply} =
        allocate(sortfreqs(Frequencies),
                 Pid),
        allocate(Frequencies, Pid),
      reply(Pid, Reply),
      loop(NewFrequencies);

...

sortfreqs({Freqs, Allocated}) ->
  {lists:sort(Freqs), Allocated}.
```

Listing 3: *Modifications for `lifo` allocation*

```
  Frequencies = {get_frequencies(), []},
  loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11].
get_frequencies() -> [10,11,12].

%%% The client Functions

stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).
```

Listing 4: *Modifications for 3 frequencies*

6. Conclusion

In this paper we have described the framework *Synapse*, which makes use of the facilities provided by *StateChum* to allow the user to compare different implementations of systems with similar behaviours. As far as we are aware this is the first such system by which a user can see the differences between inferred models. Crucially, the abstraction and simplification provided by an FSM view – rather than concrete code – can make it easier for a user to locate and understand the material differences between implementations, or discrepancies between specification and implementation.

We have illustrated some of the main features, and we have shown in practice how it can help understand the effect that different design choices have in the general behaviour of a system. These analysis tools have proven useful for visually discovering properties in the program as well as for achieving a higher confidence in that the program works as expected. The information provided by these tools can also make the choices easier when comparing different implementations with similar behaviours, and it can help for documenting purposes or as a visual aid for cooperative discussion.

Clearly, there is scope for extending this systems. For example, the current inference process doesn't incorporate data or parameters into the model. Ongoing research work into inferring EFSM models could be incorporated into the *Synapse* interface to make it easy to select between inference approaches.

Additionally, the inference process operates on a single Erlang module and produces a model whose behaviour is presented as being sequential. This is often appropriate for a high level abstraction of the API of a system, but the particular emphasis on concurrency and distributed systems in Erlang would benefit from a system that was able to capture this aspect of the system's behaviour.

Acknowledgments

This work was done as part of the EU FP7 project PROWESS, http://www.prowess-project.eu, grant no. 317820.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75:87–106, 1987. ISSN 0890-5401. . URL http://dl.acm.org/citation.cfm?id=36888.36889.

[2] T. Arts and S. J. Thompson. From test cases to FSMs: augmented test-driven development and property inference. In S. L. Fritchie and K. F. Sagonas, editors, *Erlang Workshop*, pages 1–12. ACM, 2010. ISBN 978-1-4503-0253-1.

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10. ACM, 2006.

[4] T. Arts, P. L. Seijas, and S. Thompson. Extracting QuickCheck Specifications from EUnit Test Cases. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pages 62–71. ACM, 2011. ISBN 978-1-4503-0859-5.

[5] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transaction on Computers*, 21:592–597, 1972.

[6] K. Bogdanov and N. Walkinshaw. Computing the Structural Difference between State-Based Models. In A. Zaidman, G. Antoniol, and S. Ducasse, editors, *WCRE*, pages 177–186. IEEE Computer Society, 2009. ISBN 978-0-7695-3867-9.

[7] K. Bogdanov, N. Walkinshaw, and R. Taylor. StateChum. http://statechum.sourceforge.net/ [Accessed 14th January 2014].

[8] F. Cesarini and S. Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596518188, 9780596518189.

[9] P. Dupont, B. Lambeau, C. Damas, and A. V. Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22:77–115, 2008.

[10] G. Fraser and N. Walkinshaw. Behaviourally Adequate Software Testing. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[11] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. URL http://www.isrl.uiuc.edu/~amag/langev/paper/gold67limit.html.

[12] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998. URL http://portal.acm.org/citation.cfm?id=645517.655780.

[13] K. J. Lang, B. A. Pearlmutter, and R. A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In V. Honavar and G. Slutzki, editors, *Grammatical Inference; 4th International Colloquium, ICGI-98*, volume 1433 of *LNCS/LNAI*, pages 1–12. Springer, 1998.

[14] R. Taylor, K. Bogdanov, and J. Derrick. Automatic Inference of Erlang Module Behaviour. In E. B. Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2013. ISBN 978-3-642-38612-1, 978-3-642-38613-8.

[15] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 248–257. IEEE Computer Society, 2008. . URL http://dx.doi.org/10.1109/ASE.2008.35.

[16] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2007.

[17] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont. STAMINA: a competition to encourage the development and assessment of software model inference techniques. *Empirical Software Engineering*, 18(4):791–824, 2013.

[18] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring Extended Finite State Machine models from software executions. In R. Lämmel, R. Oliveto, and R. Robbes, editors, *WCRE*, pages 301–310. IEEE, 2013.

[19] E. J. Weyuker. Assessing Test Data Adequacy through Program Inference. *ACM Transactions on Programming Languages and Systems*, 5 (4):641–655, 1983.