Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра системного программирования

Erlang. Статический и динамический анализ

Курсовая работа студентки 445 группы
Гущиной Веры Михайловны

Научный руководитель Урбанович С. Ю.
аспирант СПбГУ

Санкт-Петербург
2011 г.

Содержание

1 Введение

В реальном мире безошибочных программ не существует. Обычно ошибки выявляются при тестировании и использовании программного обеспечения (ПО) конечными пользователями. Часто это бывает либо недопустимо, либо очень дорого. Естественным желанием является перенесение выявления ошибок на более ранние этапы разработки ПО. Существует несколько основных методик, позволяющих добиться этого.

- Unit-тестирование;
- Верификация программ;
- Статический анализ;
- Динамический анализ.

Каждая из этих методик обладает своими сильными и слабыми сторонами. Хороший результат достигается лишь при использовании нескольких из них одновременно. Рассмотрим кратко каждую по отдельности.

Написание *Unit-тестов* отнимает очень много времени, а обеспечение полного покрытие кода тестами чаще всего невозможно. Однако эта практика получила широкое распространение (Test Driven Development или TDD), так как написание отдельных тестов обычно является простым, доступным каждому процессом.

Формальная верификация программ является очень трудозатратным способом нахождения ошибок, так как данный метод требует использования мощного математического аппарата. Верификатор проверяет, что программа соответствует заданным свойствам и удовлетворяет построенной математической модели. Этот метод считается очень эффективным, так как позволяет с очень большой степенью точности удостовериться, что в приложении отсутствуют ошибки.

Следующие два метода отличаются простотой для применяющего их программиста. Они не позволяют найти всех ошибок в програмном обеспечении, а нахождение большего числа ошибок напрямую связано с увеличением количества ложных срабатываний. Однако, методы позволяют находить многие типичные ошибки в ПО. Статический и динамический анализ являются перспективными методами, так как затраты по созданию подобных решений нахождения ошибок не включаются в какой-то конкретный проект. Анализаторы универсальны и могут быть применены к любой программе, написанной на языке и/или для платформы, которые поддерживаются конкретным анализатором.

Статический анализ — методика исследования программы по её исходному или бинарному коду без его исполнения. Метод дает лучшие результаты для языков с сильной статической типизацией при отсутствии (или четкой локализации) побочных эфферктов. На практике, многие применяемые языки программирования имеют динамическую типизацию, а программы зачастую пишутся в императивной парадигме. Хотя это упрощает

написание программ, это зачастую затрудняет статический анализ кода и его отладку. Для многих популярных языков уже существуют статические анализаторы различного качества, они часто применяются для таких языков, как C [1, 2], Python [3], etc.

Динамический анализ, в противоположность статическому анализу, описанному ранее, проводит исследование программы во время её исполнения. С помощью этой методики возможен поиск ошибок, которые нельзя найти другими методами. Одним из классических примеров является поиск состояний гонки (race condition) при помощи испопользования динамического анализа [4]. Главным недостатком такого типа анализа является принципиальная невозможность провести исследование кода, который при исполнении программы не выполнялся.

1.1 Постановка задачи

Целью данной работы является создание анализатора для менее популярного, но все более входящего в реальные практики программирования языка *Erlang* [5]. В первом приближении требуется понять, возможно ли создание подобного программного решения на менее сложной задаче, используя методы статического и динамического анализа. В качестве такой задачи была выбрана задача построения *графа вызовов функций* (call graph). Граф вызовов функций помогает понять, какие функции и/или модули зависят друг от друга. Эту информацию можно использовать при создании документации для разработчиков. Такую документацию сложно поддерживать в актуальном состоянии, и автоматическое построение графа вызовов позволяет частично решить эту проблему.

Язык программирования Erlang и его платформа (реализация), имеющая название *Open Telecom Platform* [5] (OTP), имеет следующие особенности, отличающие его от других языков, а значит, влияющие на реализацию анализаторов:

- легковесные процессы,
- строгая динамическая типизация,
- функциональная парадигма программирования,
- передача сообщений (message passing),
- поддержка функций первого порядка,
- широкие возможести интроспекции.

Так, если в объектно-ориентированных языках программирования основной логической единицей является класс, то в Erlang такой единицей будет процесс. В языке сохраняется баланс между чистотой (referential transparency) и простотой написания реальных приложений. Программы могут содержать аннотации типов [6], такая возможность появилась

недавно, и является необязательной по стандарту. Хотя приложения на этом языке написаны по большей части в функциональной парадигме программирования, так в языке отсутствуют циклы, язык не является чистым, но побочные эффекты хорошо локализированы.

Стоит отметить, что с помощью статического анализа отследить, куда и кем отправлялись сообщения, в каком процессе вызывались какие функции, отследить вызовы функций, переданные аргументом или возвращенные, не представляется возможным. Богатые возможности *интроспекции* (introspection) платформы Erlang/OTP позволят реализовать динамический анализатор.

1.2 Существующие решения

Проблема анализа кода и построения графа вызовов уже ставилась другими разработчиками, и на данных момент существует несколько приложений, призванных решить поставленную задачу. Самой популярной утилитой для статического анализа кода на Erlang является *dialyzer* [7], он входит в стандартную поставку платформы Erlang/OTP.

Dialyzer может осуществлять проверку на корректность по типу, учитывая аннотации, и осуществлять другие действия, в том числе строить граф вызовов функций [8]. Однако у него есть очень существенный недостаток: чтобы анализировать код, ему требуется первоначальная индексация всех модулей. Эта индексация порой может занимать несколько суток, и, хотя на Erlang достаточно просто писать приложения, использующие несколько процессоров/ядер или даже несколько разных физических машин (кластер), dialyzer может использовать только один процессор для индексации модулей.

Вторым существенным недостатком этого решения является невозможность осуществлять динамический анализ программы. Как уже было отмечено ранее, с помощью только статического анализа невозможно составить полный граф вызовов функций, т.к. они могут быть вызваны неявно, например, с помощью функции `erlang:apply/3`.

Виртуальная машина Erlang/OTP имеет богатые возможности интроспекции кода. Чтобы вопрользоваться этой функциональностью, разработчики обычно пишут пару сотен строк дополнительного кода, решающего конкретную задачу. Хотя метод является широкоиспользуемым, он, очевдино, имеет недостатки: так, он не учитывает информацию, которую можно получить с помощью статического анализа, а код интроспектора дублируется от проекта к проекту [9].

2 Статический анализ

Стандартным подходом для реализации статического анализатора является разбор исходного кода программы в *абстрактное синтаксическое дерево* (Abstract Syntax Tree, AST), над которым осуществляется дальнейший анализ [10].

Особенностью реализации исполнения Erlang-кода на платформе Erlang/OTP является использование виртуальной машины Bogdan/Björn's Erlang Abstract Machine (BEAM): перед исполнением исходный код программы компилируется в байт-код виртуальной машины. При указании опции компилятора `+debug_info` в байт-код включается соответствующее ему дерево разбора. Данная опция практически всегда применяется при компиляции исходного кода (так, все библиотеки скомпилированы с указанием этой опции), так как она позволяет проводить интроспекцию кода во время исполнения. Используя абстрактное синтаксическое дерево, включенное в байт-код, можно решить ряд проблем:

- не требуется реализовывать поиск синтаксических ошибок исходного кода программы, т.к. анализ на синтаксическую корректность уже проведен компилятором в байт-код;
- не требуется иметь доступ к исходным кодам программы, что актуально для стандартных прекомпилированных модулей;
- абстракное синтаксическое дерево из байт-кода всегда соответствует байт-коду исполняемой программы, что актуально при реальной разработке.

В связи с вышеперечисленными преимуществами было принято решение воспользоваться деревом разбора, доступным в beam-файлах. Для получения этого дерева существует библиотека *beam_lib* [11], входящая в стандартную поставку Erlang/OTP. Решение использовать разбор beam-файлов определило язык реализации синтаксического анализатора: он был реализован на самом Erlang.

```
1> beam_lib:chunks("appmon.beam", [abstract_code]).
{ok, {appmon,
        [{abstract_code,
           {raw_abstract_v1,
             [{attribute, 1, file, {"appmon/src/appmon.erl", 1}},
              {attribute, 18, module, appmon},
              {attribute, 19, behaviour, gen_server},
              {...}
              {function, 75, start, 0, [{clause, 75, ...}]},
              {function, 78, stop, 0, [{clause , ...}]},
              {function, 93, init, 1, [{...}]},
              {...}|...]}}]}}
```

Пример 2.1: Результат работы функции `beam_lib:chunks/2`

Одной из функций библиотеки beam_lib является функция `beam_lib:chunks/2`, в качестве параметров которой передается: первым аргументом имя модуля, имя файла или его содержимое, а вторым аргументом список Erlang-термов, определяющих, какую информацию требуется извлечь из указанного модуля. Результатом выполнения функции является список термов эрланга. В примере 2.1 показан результат работы функции `beam_lib:chunks/2`.

```
[{'appmon:init/1', 'appmon:display_setopt/2'},
 {'appmon:init/1', 'appmon:draw_win/2'},
 {'appmon:init/1', 'appmon:mk_mnodes/2'},
 {'appmon:parse_apps/2', 'appmon:parse_apps/2'},
 {'appmon:parse_nodes/1', 'appmon:parse_nodes/2'},
 {'appmon:parse_nodes/2', 'appmon:parse_apps/2'},
 {'appmon:parse_nodes/2', 'appmon:parse_nodes/2'},
 {...}|...]
```

Пример 2.2: Вывод статического анализатора графа вызовов функций

Синтаксический анализатор, принимая на вход список термов, возвращает граф вызовов функций в виде списка пар вида вызывающая – вызываемая функции. Результат работы анализатора для списка термов из примера 2.1 представлен в примере 2.2.

Код статического анализатора получился очень лаконичным и простым благодаря использованию концепции *сопоставления по шаблону* (pattern matching), доступной в языке Erlang. Хотя виртуальная машина BEAM отличается медленной работой с арифметическими операциями, по документации код на Erlang медленнее на порядок [12] аналогичного кода, написанного на языке C, полученный статический анализатор показывает высокую производительность на реальных данных, так как он использует, как было сказано выше, в основном сопоставление по шаблону. Pattern matching является одной из базовых концепций Erlang, и производительность программ, использующих данный прием программирования, близка к производительности аналогичных программ, написанных на C. В примере 2.3 представлена небольшая часть кода, реализующая статический анализ через обход абстрактного синтаксического дерева:

```
...
analyze(F, {cons, _, H, T}) ->
    analyze(F, H) ++ analyze(F, T);
analyze(F, {generate, _, _Var, Generator}) ->
    analyze(F, Generator);
analyze(F, {call, _, {remote, _, {atom, _, Module},
            {atom, _, Funcname}}, Params}
) -> [{F, {Module, Funcame}} | analyze(F,Params)];
analyze(F, {call, _, {atom, _, Funcname}, Params}) ->
    [{F, Funcname} | analyze(F, Params)];
analyze(F, {clause, _, _, _, Code}) ->
    analyze(F, Code);
...
```

Пример 2.3: Часть кода, реализующая статический анализ

3 Динамический анализ

Как было сказано ранее в параграфе 1.1, с помощью лишь статического анализа невозможно получить всю необходимую информацию. Недостающую информацию возможно получить, используя динамический анализ. Его можно осуществить, используя возможность интроспекции платформы Erlang/OTP. Чтобы воспользоваться данной функциональностью, существует набор функций, из которых в данной работе использовались следующие [13]:

- `erlang:trace(PidSpec, How, FlagList)`;
- `erlang:trace_pattern(MFA, MatchSpec, FlagList)`.

С помощью первой функции, `erlang:trace/3`, можно указать, за какими процессами и за какими действиями требуется вести наблюдение. Вторая функция позволяет определить, какие модули и/или функции, вызывающиеся в рамках наблюдаемых процессов, нужно отслеживать. Сообщения о происходящих *событиях* времени выполнения будут посылаться процессу, вызвавшему функцию `erlang:trace/3`.

В примере 3.1 показана простейшая программа на языке Erlang, которая логирует все вызовы фукций во всех процессах, созданных после запуска логирующего процесса:

```
tracer() ->
    erlang:trace(new, true, [call]),
    erlang:trace_pattern({'_', '_', '_'}, true),
    loop().

loop() ->
    receive
        {trace, Pid, call, {M, F, Args}} ->
            io:format("~p: ~p~n", [Pid, {M, F, Args}]);
        Other ->
            io:format("Error: ~p~n", [Other])
    end,
    loop().
```

Пример 3.1: Простейший процесс, логирующий все вызовы функций

Механизм передачи сообщений является единственным способом взаимодействия процессов Erlang. Этот механизм действует как внутри данного процесса операционной системы, так и между Erlang-процессами, запущенными на физически разных хостах.

С помощью динамического анализатора в данном случае можно получить два типа дополнительной информации, интересной для дальнейшего исследования:

- вызов функции неявно (эту информацию нельзя получить с помощью статического анализа);

- информация о выполняющихся процессах (как было отмечено выше, эта информация очень важна и ее также нельзя получить с помощью статического анализа. В Erlang, в отличие от объектно-ориентированных языков, единицей абстракции выступают именно процессы, см. 1.1).

Отличительной особенностью динамического анализатора является тот факт, что информация поступает и обновляется в потоковом режиме, а сообщения о событиях приходят с минимальной задержкой в режиме мягкого *реального времени* (soft real-time). Таким образом, появляется возможность наблюдать за состоянием виртуальной машины Erlang и поддерживать информацию о выполнении в актуальном состоянии. В свою очередь такой механизм накладывает и ограничение: анализатор обязан реагировать на поступающие события как можно быстрее, что ограничивает сложность анализа в режиме по требованию.

Информацию о косвенных вызовах функций можно добавлять к уже построенному на этапе статического анализа графу вызовов функций. На этом этапе также возможно отследить, какие функции существуют в коде, но никогда не исполняются, что позволит найти мертвый код. Этот факт очень полезен при дальнейшем рефакторинге кода программы. Информацию о процессах требуется визуализировать отдельно и желательно в потоковом режиме.

При реализации динамического анализатора были написаны обработчики следующих собитый:

- вызов функций от имени процесса,
- создание, завершение и соединение процессов,
- посылка и принятие сообщений.

Для облегчения совмещения информации о графе вызовов функций было логично выбрать тот же самый тип выходных данных для динамического анализатора, что использовался в статическом анализаторе: список пар вида вызываемая – вызывающая функции (см. главу 2).

4 Визуализация результата

Так как объем полученной информации от статического и динамического анализаторов является большим, и между данными существует много ссылок друг на друга, эту информацию очень трудно воспринимать в чисто-текстовом виде, поэтому была поставлена задача визуализации получаемых данных.

4.1 Визуализация графа вызовов функций

Дерево вызовов, а также отношения между процессами виртуальной машины Erlang логичнее всего визуализируются в виде плоского графа. Задача визуализации графа очень популярна, существует множество средств, позволяющих решить эту проблему [14]. Одной из самых популярных утилит в этой области является утилита GraphViz [15]. В качестве входных данных GraphViz принимает файл в формате *dot* [16]. Для того, чтобы визуализировать граф вызовов, полученный от статического анализатора, был написан сериализатор из списков пар вида вызываемая – вызывающая функции в формат dot.

Стоит отметить, что как вызываемая, так и вызывающая функции состоят из имени самой функции и имени модуля, в котором эта функция определена. Таким образом, на отображаемом графе можно построить еще одну градацию отношений — зависимость модулей. Пример работы визуализатора данных с использованием GraphViz представлен на рисунке 4.1.

Как было отмечено в главе 3, существует два типа информации, интересной для дальнейшего исследования, получаемой от динамического анализатора:

- информация о динамических вызовах функций,
- данные о взаимосвязи процессов в виртуальной машине Erlang.

И если первый тип не требует отображения в реальном времени, так как эта информация не даёт никаких конкурентных преимуществ и лишь усложняет понимание графа вызовов, то для отображения второго типа данных требуется визуализация с учетом времени возникновения событий.

4.2 Потоковая визуализация

Для визуализации взаимодействия процессов виртуальной машины Erlang было решено воспользоваться приложением Gephi [17], которое позволяет визуализиваровать данные в реальном времени. На момент написания курсовой работы в стабильной версии данного приложения 0.7 отсутствовала возможность визуализации графов с учетом времени возникновения событий. Андре Паниссон в рамках программы Google Summer of Code 2010

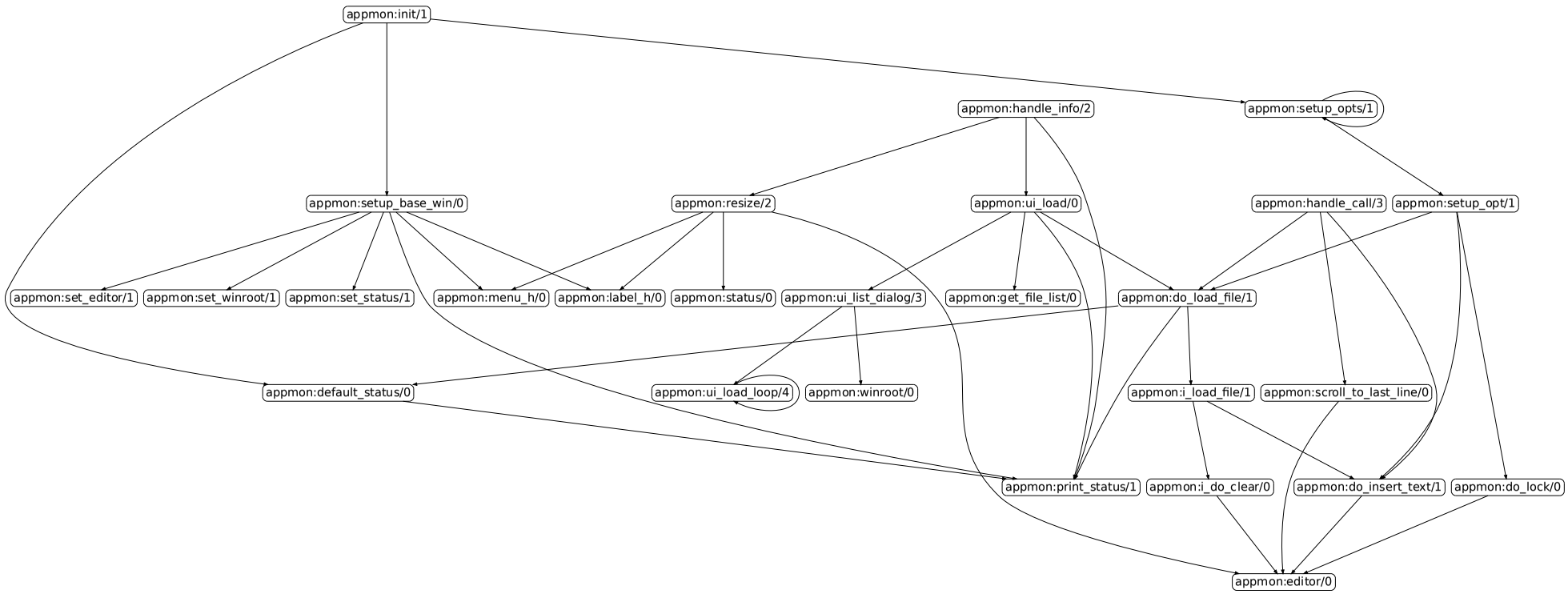

Рис. 4.1: Визуализация графа вызова функций с использованием GraphViz

реализовал в Gephi поддержку потокового интерфейса обновления информации о графе [18] (*Graph Streaming API*), с помощью которого возможна визуализация данных с учетом времени возникновения событий, эта функциональность доступна в Gephi начиная с версии 0.8-alpha. Чтобы воспользоваться данным интерфейсом приложения, для динамического анализатора был написан простейший веб-сервер, отдающий данные, сериализованные в формате JSON [19], используя передачу данных по протоколу HTTP по частям [20] *по мере поступления информации*. Результат работы визуализатора данных о процессах представлен в виде видеозаписи в приложении к курсовой. На рисунке 4.2 представлено статическое изображение полученного результата, из-за большого количества текстовой информации на нем отсутствуют подписи.

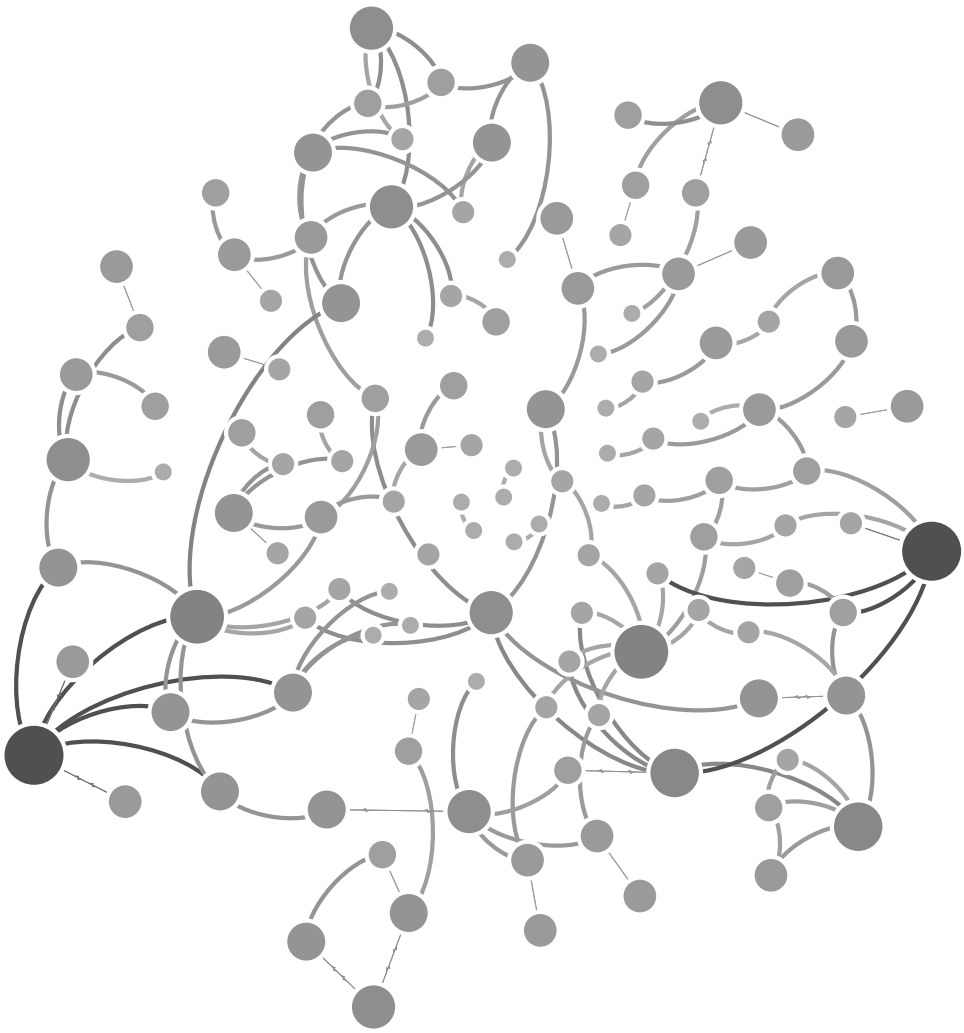

Рис. 4.2: Визуализация связей между процессами

5 Заключение

Задача, поставленая при написании курсовой работы, была решена. Удалось построить граф вызовов функций, используя статический анализатор для Erlang и дополнив его информацией от динамического анализатора. Дополнительным результатом можно считать возможность получения информации об исполняющихся процессах и взаимосвязях между ними. Также были реализованы визуализаторы получаемых данных, в том числе удалось визуализировать данные в реальном времени, что позволяет лучше понимать процессы, происходящие в виртуальной машине Erlang/OTP. Производительность полученного программного обеспечения, в отличие от конкурирующих решений, не вызывала нареканий.

Список литературы

[1] *Clang*. http://clang.llvm.org/.

[2] *Sparse*. https://sparse.wiki.kernel.org/index.php/Main_Page.

[3] *PyLint*. http://pypi.python.org/pypi/pylint.

[4] *ThreadSanitizer*. http://code.google.com/p/data-race-test/.

[5] *Erlang Programming Language*. http://erlang.org/.

[6] *EEP 8: Types and function specifications*. http://erlang.org/eeps/eep-0008.html.

[7] Konstantinos Sagonas *Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications*. Bugs'05. http://user.it.uu.se/~kostis/Papers/bugs05.pdf.

[8] *Dialyzer Reference Manual*. http://www.erlang.org/doc/man/dialyzer.html.

[9] Максим Трескин *Инструменты интроспекции в Erlang/OTP*. Практика функционального программирования, Выпуск 5. ISSN 2075-8456.

[10] Aho, A. V., Sethi, R., Ullman, J. D., Lam, M. S. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2006. ISBN 0-321-48681-1.

[11] *STDLIB Reference Manual – beam_lib*. http://www.erlang.org/doc/man/beam_lib.html.

[12] *Frequently Asked Questions about Erlang – How do I...* http://www.erlang.org/faq/how_do_i.html#id49439.

[13] *Erlang Run-Time System Application (ERTS) Reference Manual*. http://www.erlang.org/doc/man/erlang.html.

[14] Tollis, I. G., Battista, D. G., Eades, P., Tamassia, R. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998. ISBN: 0-133-01615-3.

[15] *GraphViz – Graph Visualization Sotfware*. http://graphviz.org/.

[16] *The DOT Language*. http://www.graphviz.org/doc/info/lang.html.

[17] *Gephi, an open source graph visualization and manipulation software*. http://gephi.org/.

[18] *Specification – GSoC Graph Streaming API.* http://wiki.gephi.org/index.php/Specification_-_GSoC_Graph_Streaming_API.

[19] *The application/json Media Type for JavaScript Object Notation (JSON).* http://www.ietf.org/rfc/rfc4627.txt.

[20] *Hypertext Transfer Protocol – HTTP/1.1.* http://www.ietf.org/rfc/rfc2616.txt.