Статический анализ РуСи

Меньшиков М.А., аспирант кафедры системного программирования
СПбГУ, info [at] menshikov.org

Аннотация

Язык РуСи является безопасной альтернативой C благодаря использованию некоторого поднабора операций. Программы на этом языке компилируются в коды виртуальной машины, которая, в свою очередь, выполняет дополнительные проверки, способные предотвратить незапланированное поведение. Однако контроль корректности может осуществляться ещё до компиляции — путем статического анализа. В данной работе описывается интеграция статического анализатора в проект РуСи, классы поддерживаемых им ошибок, оформление аннотаций, а также очерчиваются перспективы дальнейшего развития инструмента.

Введение

Язык РуСи первоначально был предложен А.Н. Тереховым как средство обучения программированию [1] на базе роботов ТРИК. Будучи синтаксически основанным на C, в нем существенно сужается круг доступных небезопасных конструкций. К примеру, отсутствует арифметика указателей. Другой особенностью проекта является применение виртуальной машины, которая может выполнять различные проверки безопасности действий. Такие ошибки, как выход за границы массива, деление на ноль, переполнение стека, некорректные аргументы функций — успешно обнаруживаются ещё до момента, когда они действительно могут причинить ущерб.

Альтернативой проверкам в реальном времени может служить статический анализ. Идеологически, он способствует двум целям, адаптация каждой из которых ведет к объективно более надежным программам.

1. *Доказательство корректности программ*. Не все виды ошибок могут эффективно выявляться в реальном времени: некоторые из них могут проявляться в специфичных условиях. Такие ситуации можно обнаружить исследованием пространства состояний и сравнением кода с его спецификацией.

2. *Документирование кода*. Код с качественно написанной спецификацией — объясним, читаем и нагляден.

Проект статического анализатора начался с экспериментов в области поиска причин состояний гонок в ядре Linux, и постепенно трансформировался в полноценный многоязычный инструмент, опционально интегрирующийся с компилятором РуСи.

Классы поддерживаемых проблем

Статический анализатор обнаруживает следующие проблемы — в порядке от простых к сложным:

1. **Синтаксические ошибки**. Так как анализатор основывается на Clang, то автоматически обнаруживаются несоответствия стандарту C. Также обнаруживаются простые ошибки вроде одинаковых операндов в операции сравнения.

2. **Арифметические и логические ошибки**. Всегда выполняющиеся и всегда не выполняющиеся условия, непреднамеренная потеря информации при проведении побитовых операций, переполнения и другие ошибки.

3. **Ошибки доступа к памяти**. Выход за пределы массивов, использование неинициализированных данных, разыменование нулевых указателей.

Для обнаружения данных ошибок используется комбинация методов *проверки модели* и *абстрактной интерпретации*. Статьи непосредственно об алгоритмах в настоящий момент готовятся, поэтому логичным будет опустить описание алгоритмов, но подчеркнуть особенности работы анализатора с РуСи.

Особенности работы анализатора с РуСи

Парсинг РуСи в проекте организован достаточно просто в связи с его совместимостью с C. Русскоязычные ключевые слова на ранней стадии заменяются на английские варианты, и далее за процесс отвечает Clang. При этом никакой действительной привязки к API этого анализатора нет, так как парсинг происходит по схеме *исходный код* $\rightarrow$ *Clang AST* $\rightarrow$ *обобщенные синтаксические деревья*. Последняя сущность — нововведение анализатора, общее дерево для всех возможных языков программирования. Ход дальнейших действий частично описан в статье [2], но в контексте РуСи более интересны решения, касающиеся непосредственно этого языка.

В процессе дальнейшего анализа есть четыре существенных отличия по сравнению с C. Во-первых, для каждого языка вводится свой класс, описывающий возможности. При обработке каждого выражения анализатор проверяет *допустимость* операции в соответствии с требованиями языка. Если «черный список» для C пуст, то для РуСи определено множество запретов: работа с указателями, применение объединений и т.п. Появление подобных конструкций вызывает *синтаксическую ошибку*, которая на этапе парсинга через Clang, по понятным причинам, не обнаруживается.

Во-вторых, каждый из встроенных решателей (Model Checker, абстрактный интерпретатор) генерирует дополнительные ограничения на модель программы. Примером[1] может служить простейшая проверка длины массива по указателю:

```
int  buf[5];
int *b = buf;
b[7] = 5;
```

С точки зрения C, здесь может и не быть ошибки, так как возможно обращение ко всему сегменту данных, поэтому проверка выхода за границы массивов требует некоторых эвристик и, по большей части, опциональна. Большая строгость РуСи заставляет явным образом сгенерировать проверку $7 < arraylen(buf)$.

В-третьих, стандартная библиотека РуСи принципиально отличается от таковой в C. К примеру, `scanf` и `printf` принимают другие маски типов данных. Присутствуют не имеющие аналогов функции работы с датчиками, WiFi и т.п., каждая из которых требует собственной спецификации.

В-четвертых, присутствуют зачатки синтаксического сахара [3] для более удобной работы с потоками — `t_create_direct`, `t_exit`. Для поддержки возможной реализации, была проведена работа по унификации представлений функций.

Интеграция в РуСи

Компилятор RuC состоит из парсера и генератора кода, и он не строит какую-либо модель программы. По этой причине интеграция анализатора напрямую в код достаточно проблематична. Между тем, анализатор должен выявлять ошибки на разных уровнях — абстрактное синтаксическое дерево, граф потока исполнения, логические ошибки и т.п.

[1] Несмотря на то, что в статье приводятся простые примеры, они важны для иллюстрирования *важных формальных* отличий языков.

Анализатор реализован как отдельный инструмент. В настоящий момент разрабатывается легковесная библиотека, организующая доступ к основным API посредством JSON-RPC по сети, через pipes, unix-сокеты и т.п. К примеру, простейшие запросы на анализ файла `test.c` выглядят следующим образом:

```
{ "jsonrpc": "2.0", "method": "workspace_create",
  "params": { "files": [ "/home/user/test.c" ] },
  "id": 1
}
{ "jsonrpc": "2.0", "method": "analyze_file",
  "params": { "workspace_id": 0, "file": 0,
              "language": "ruc" },
  "id": 2
}
```

В ответ может прийти примерно такой ответ:

```
{ "jsonrpc": "2.0",
  "result": {
    "report": [ { "line": 5, "column": 36,
                  "message": "Precondition is violated",
                    "contract": "x >= 0",
                    "function": "int f(int x)",
                    "line_content": "res = f(-1);",
                    "author": "user",
                    ... } ] },
  "id": 2
}
```

Такой подход решает несколько проблем:

1. **Обобщение API** (со стороны компилятора и анализатора). Нет необходимости переделывать плагин компилятора при каждом значимом изменении внутреннего API. К тому же, это приводит к разработке анализатора с фокусом на потребности пользователя, выражающиеся в лаконичном API.

2. **Уменьшение числа зависимостей, ускорение разработки** (со стороны компилятора). Анализатор сравнительно большой, несет значительное число зависимостей (LLVM/Clang, CVC4, antlr4, asio, mongocxx-driver, ...), но физическое разделение этих проектов позволяет им развиваться независимо.

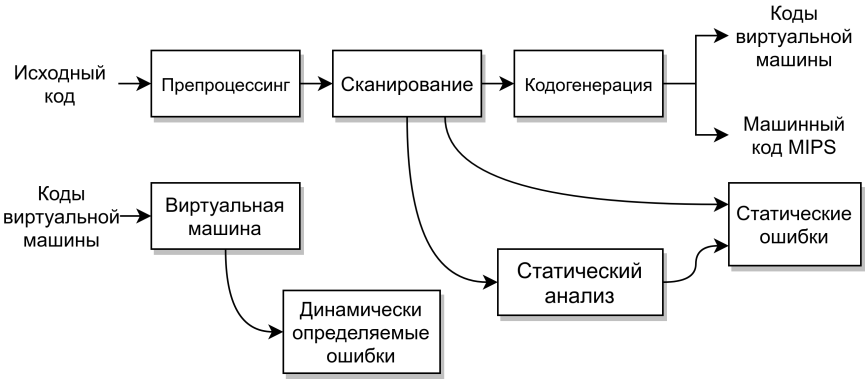

Рис. 1: Схема интеграции анализатора в инфраструктуру РуСи

3. **Реализация сервера статического анализа для различных приложений** (со стороны анализатора). Достаточно правильным выглядит решение разрабатывать унифицированный интерфейс для разных приложений. В дальнейшем это может помочь развернуть полноценный графический интерфейс.

Схема интеграции приведена на рисунке 1.

Аннотации

Для описания спецификаций функций используется язык ANSI/C Specification Language [4] (ACSL), введенный авторами анализатора Frama-C [5]. Основными преимуществами такого описания являются *простота*, *выразительность*, также такие аннотации применимы в качестве документации благодаря чистоте представления.

Анализатором поддерживаются предусловия `requires`, постусловия `ensures`, статические проверки `assert`, инварианты циклов `loop invariant`, именованные поведения и другие элементы ACSL. Большая часть их часть применяется *перед* определением или объявлением функции, а статические проверки можно применять в любом месте внутри функции.

Аннотации обеспечивают лишь некоторые проверки безопасности, их вполне может не хватать для полноценной верификации, но зачастую для поиска *реальных* проблем достаточно и такого варианта. Часть проверок встроена в анализатор и полностью невидима для пользователя.

Приведем простой пример аннотирования функции:

```
/*@
  @ divisor_not_zero: requires divisor != 0;
  @                   ensures \result == (num / divisor);
  */
цел div(цел num, цел divisor)
{
    возврат num / divisor;
}
```

Здесь вводится именованное поведение `divisor_not_zero` (неравенство делителя нулю) с предусловиями и соответствующими постусловиями.

Дальнейшее развитие анализа РуСи

В дальнейшем планируется углубить анализ программ на РуСи. Одно из направлений — улучшение самого анализатора. Непрерывно ведется работа над улучшением реализаций методов проверки модели и абстрактной интерпретации, используемых в проекте. Добавляется *распределенность* [2], улучшается API для доступа к возможностям анализатора.

Другое направление связано непосредственно с усилением интеграции в компилятор РуСи. Постепенно, с его модернизацией, возможно встраивание вызовов API напрямую, минуя JSON-RPC. Это позволит поспевать за изменениями семантики языка, а также плотнее интегрироваться в систему вывода сообщений об ошибках. Возможна автоматическая генерация библиотеки спецификаций по реализации интерпретатора РуСи. В будущем также желательно введение аннотаций в синтаксис языка.

Заключение

В работе рассмотрены предпосылки разработки статического анализатора РуСи, связанные с необходимостью обнаруживать ошибки до их непосредственного детектирования виртуальной машиной РуСи. Описаны классы поддерживаемых ошибок, приведен способ интеграции анализатора в рабочую среду компилятора. Рассмотрены аннотации и приведен пример их использования. Очерчены перспективы дальнейшего развития анализатора кода.

Литература

[1] Терехов А.Н. Инструментальное средство обучения программированию и технике трансляции // КИО. 2016. №1.

[2] Menshchikov M. Scalable semantic virtual machine framework for language-agnostic static analysis // Proceedings of The 8th International Conference ”Distributed Computing and Grid-technologies in Science and Education” (GRID). 2018. P. 213–217.

[3] Терехов А.Н., Головань А.А., Терехов М.А. Параллельные программы в проекте РуСи // КИО. 2018. №2.

[4] Baudin P. et al. ACSL: ANSI C Specification Language. – 2008.

[5] Cuoq P. et al. Frama-C // International Conference on Software Engineering and Formal Methods. – Springer, Berlin, Heidelberg, 2012. – P. 233-247.