Формальная верификация многопоточного скриптового окружения

Никольский К. А., студент кафедры системного программирования СПбГУ,
nikolskiy.kirill.andreevich@gmail.com

Мордвинов Д. А., аспирант кафедры системного программирования СПбГУ,
mordvinov.dmitry@gmail.com

Аннотация

В данной работе была проведена формальная верификация многопоточного скриптового приложения методом *проверки модели* (*model checking*). При построении модели были смоделированы компоненты и примитивы инструментария Qt[1], с помощью которого была написана исходная программа. На момент написания статьи верификатором SPIN [1] были найдены восемь ошибок в исходной программе.

Введение

Компьютерные программы часто содержат ошибки. Еще Эдсгер Вибе Дейкстра говорил, что тестирование не позволяет обнаружить все ошибки. Но для автоматических систем критически важна корректная и безотказная работа всех частей.

Таким образом, *формальная верификация* программного кода, позволяющая существенно повысить качество системы, становится важнейшей областью научных исследований в информатике. Формальной верификацией программы, согласно [2], будем называть приемы и методы формального доказательства (или опровержения) того, что модель программной системы удовлетворяет заданной формальной спецификации (см. рис. 1). Важно понимать, что проведя формальную верификацию и тестирование, нельзя гарантировать полное отсутствие ошибок в системе. Но при использовании этих методов вместе существенно повышается уровень доверия к системе.

На сегодняшний день для верификации сложных параллельных, многопоточных систем зарекомендовал себя метод проверки модели (model checking) [2]. Это метод проверки того, что данная формальная модель системы удовлетворяет ограничению, поставленному чаще всего в терминах какой-либо *темпоральной логики*, то есть логики, в которой истинность логических

[1] Qt — кроссплатформенный инструментарий разработки ПО на языке программирования C++. URL: http://www.qt.io/

утверждений зависит от времени. При таком подходе построение адекватной модели является трудоемким процессом, при котором постоянно приходится решать проблему комбинаторного взрыва.

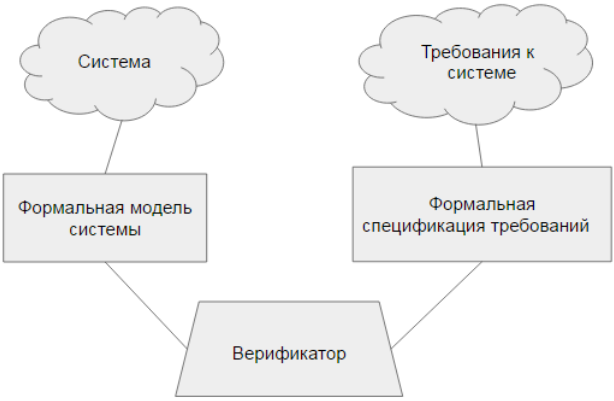

Рис. 1: Общая схема верификации

В рамках Межвузовской Проектной Лаборатории Робототехники создают программную часть для контроллера робототехнического конструктора TRIK, *окружение времени выполнения*, которая позволяет программировать робота, используя язык C++ или JavaScript. Сама программа написана на языке C++ с помощью инструментария Qt. Скриптовый язык упрощает программирование роботов, однако окружение времени выполнения является многопоточной, параллельной программой, что обуславливает высокую сложность обнаружения ошибок.

В рамках данной работы была проведена формальная верификация важной подсистемы окружения методом проверки модели. Была построена документированная, адекватная и корректная модель, что подтверждают найденные верификатором ошибки в реальной модели. Такие ошибки находились как при проверках свойств системы, выраженных в темпоральных формулах, так и при проверках внутренних ограничений модели.

Модель системы

В качестве верификатора был выбран инструмент SPIN, поскольку этот инструмент поддерживает удобную синхронизацию процессов посредством

рандеву-каналов, а также он имеет средство для моделирования асинхронного взаимодействия — асинхронные каналы с буфером. Свойства моделируемой системы будут выражаться на языке LTL[2], а также с использованием *внутренних проверок* (*asserts*)[3]. Стоит отметить, что язык Promela, входной язык для верификатора SPIN, позволяет описывать лишь конечные модели.

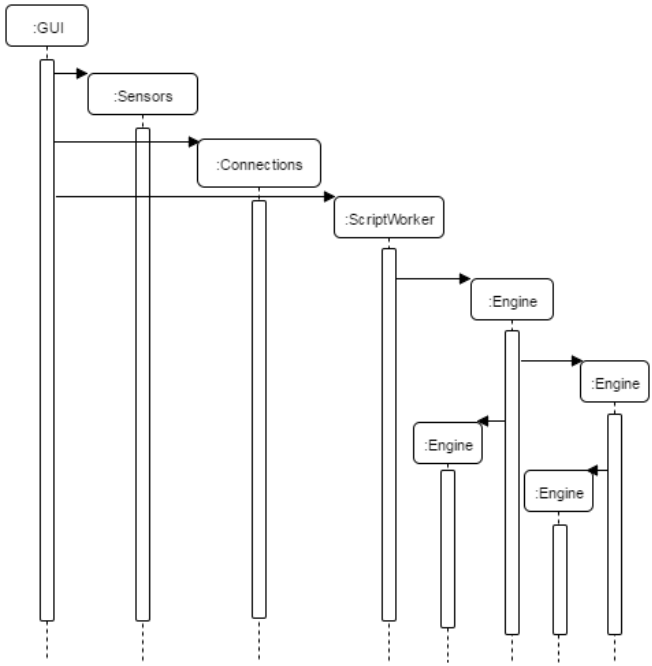

Рис. 2: Упрощенная диаграмма последовательности моделируемой системы, каждый из потоков Sensors и Connections на самом деле объединяет в себе группы потоков

Для верификации была выбрана компонента окружения времени выполнения, которая, по эмпирическим наблюдениям, была наиболее подвержена сложно воспроизводимым ошибкам. Эта компонента отвечает за исполнение пользовательских скриптов (см. рис. 2). Содержание скриптов заранее неизвестно, и, помимо прочего, они позволяют работать с потоками, создавая и завершая потоки в процессе исполнения скриптов. В данный момент, документированная, корректная и адекватная модель такой компоненты находится в процессе создания.

[2]Linear Temporal Logic [3]
[3]URL: http://spinroot.com/spin/Man/assert.html

Корректность и адекватность модели проверялись эмпирически. Один из способов оценки заключался в жуналировании различных этапов работы системы и проверке соответствия получаемых сообщений. Второй способ заключался в использовании динамического верификатора ThreadSanitizer[4], обнаруживающего гонки и потенциальные тупики в ходе работы программы. Выдаваемые таким инструментом сообщения, относящиеся к моделируемой системе, были использованы в качестве дополнительной информации для создания корректной и адекватной модели.

Процесс создания модели разбивался на анализ исходной программы и моделирование работы отдельных компонентов и примитивов инструментария Qt:

- потоки,
- очереди событий,
- система сигналов-слотов,
- обработчики исключений,
- мьютексы.

На данном этапе работы мы абстрагировались в моделируемой компоненте от потоков, отвечающих за работу сенсоров и соединений. Также пока что не смоделированы функции для вызова из пользовательских скриптов, отвечающие за коммуникацию между потоками. Таким образом, мы уменьшаем число возможных состояний моделируемой системы, что позволяет нам эффективно верифицировать модель на текущий момент.

Поток в нашей модели представляется процессом с бесконечным циклом обработки очереди событий, уникальной для каждого потока и по сути являющейся каналом с ограниченным буфером. Ограниченность буфера является проблемой, которая при верификации на данный момент решается установкой меток заключительного состояния (*ends*)[5].

Система сигналов-слотов в зависимости от типа соединения моделируется или с помощью посылок сообщений в канал необходимого потока, или простым исполнением кода в текущем потоке (последнее легко моделируется, так как почти все переменные в нашей модели являются глобальными). Таким образом, установленные соединения в исходной программе нужны лишь для определения их типов и нахождения потоков-получателей (за исключением некоторых проверок, связанных с местом объявления соединения). Все

[4] URL: https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual
[5] URL: http://spinroot.com/spin/Man/end.html

возможные сигналы описываются с помощью перечисляемого типа `mtype` и, так как заранее известны для каждого потока все соединения и сигналы, которые этот поток может обрабатывать, в нем и моделируется обработка *всех* возможных сигналов. Если необходимо передавать информацию, кроме названия сигнала, а это случается в нашей программе редко, то моделирование параметра происходит через глобальные переменные.

Обработчик исключения, опуская детали, является в модели отдельным потоком, который может принимать управление и возвращать его в нужную точку модели.

Для удобства моделирования часто используемый код выносится в так называемые (*inlines*)[6] с "говорящими" именами, и часто такой код моделирует некоторые функции исходной программы целиком.

В качестве подробного примера приведем моделирование мьютексов в рамках данной работы (удалена сопроводительная документация):

```
#define N 256
#define mutex bit
#define MutexCount 2
#define _mResetMutex 0
#define _mThreadsMutex 1
mutex mResetMutex = 1;
mutex mThreadsMutex = 1;
typedef mutexes
{
  bool forThread[N] = 1;
};
mutexes mutexInfo[MutexCount];

inline lock(_s, __s, _thread)
{
  atomic {
    assert(mutexInfo[__s].forThread[_thread]);
    _s == 1 ->
    mutexInfo[__s].forThread[_thread] = 0;
    _s--;
  };
}
inline unlock(_s, __s, _thread)
{
  atomic {
    assert(_s == 0);
    assert(!mutexInfo[__s].forThread[_thread]);
    mutexInfo[__s].forThread[_thread] = 1;
    _s++;
  };
}
```

[6] http://spinroot.com/spin/Man/inline.html

```
// Использование:
lock(mThreadsMutex, _mThreadsMutex, _pid);
unlock(mThreadsMutex, _mThreadsMutex, _pid);
```

Стоит отметить, что побочным результатом моделирования стали несколько обнаруженных ошибок и недочетов в программе, которые явно проявились именно при построении модели.

Верификация и анализ контрпримеров

Для верификации использовались внутренние проверки *asserts* и LTL-формулы. На данном этапе с помощью LTL-формул были проверены несколько свойств, касающихся корректного завершения программы и выполнимости отдельных событий. Проверяемые свойства имели следующую структуру на языке LTL:

$$\Box(p \Rightarrow \Diamond q \vee \Diamond\Box r)$$

$$\Box\Diamond p$$

$$\Diamond\Box p$$

На текущий момент модель активно разрабатывается и модифицируется, и доказано лишь одно простое свойство корректности исполнения пустого скрипта и достижения конечного состояния.

Контрпримеры анализировались и проверялись на реальной системе. В случае, если модель была неадекватной, она модифицировалась. Если проверяемое свойство нарушалось на исходной системе, ошибка записывалась в систему отслеживания ошибок. Таким образом, было найдено восемь критических ошибок, одна из которых является активной блокировкой.

Заключение

Формальная верификация в нашем случае уже позволила обнаружить восемь критических ошибок в исходной программе окружения времени выполнения, что позволяет говорить об адекватности и корректности построенной в рамках работы документированной модели. Смоделированные компоненты и примитивы инструментария Qt могут быть использованы в других моделях. На данный момент рано говорить о проверке существенных свойств системы, так как модель дорабатывается, и ещё остались непроверенными десятки контрпримеров.

Литература

[1] Holzmann Gerard J. The model checker SPIN // IEEE Transactions on software engineering. — 1997. — Vol. 23, no. 5. — P. 279.

[2] Карпов Ю. Г. Model Checking. Верификация параллельных и распределенных программных систем. — СПб.: БХВ-Петербург, 2010.

[3] Pnueli Amir. The temporal logic of programs // Foundations of Computer Science, 1977., 18th Annual Symposium on / IEEE. — 1977. — P. 46–57.