# Использование методов модельно-ориентированной обратной разработки для анализа ARINC 653 совместимого функционального ПО

С.Л. Лесовой

Институт системного программирования
им. В.П. Иванникова РАН
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
Тел. +7(495)912-56-59 (доб. 405)
lesovoy@ispras.ru

**Abstract.** This paper is dedicated to methods of Model-Driven Reverse Engineering for avionics software analysis. An algorithm for building architecture models from source code of application software is described in this paper. Architecture Analysis & Design Language is used as a target language for architecture modeling. We consider ARINC 653-compatible application software for models building. The algorithm implementation is based on CPAchecker static analyzer.

**Аннотация.** Исследуется применение методов модельно-ориентированной обратной разработки для анализа программного обеспечения (ПО) для авионики. Описывается алгоритм построения архитектурных моделей по исходному коду функционального ПО. Для описания архитектурных моделей используется язык AADL (Architecture Analysis & Design Language). Для построения моделей рассматривается ARINC-653 совместимое ПО. Предложенный алгоритм реализован на базе инструмента статического анализа CPAchecker.

## 1 Введение

Представление архитектуры системы в виде формализованных архитектурных моделей позволяет исследовать ее различные статические и динамические характеристики с помощью существующих инструментальных средств. При эксплуатации программно-аппаратных систем часто возникают задачи по их рефакторингу связанные с обновлением отдельных компонентов системы, интеграцией в существующую систему новых компонентов, переносом существующего ПО на другую аппаратную платформу и т.п. В этом случае архитектура системы строится не «с нуля», а на основе уже имеющихся программно-аппаратных компонентов. Задача построения архитектурной модели на основе имеющегося программного кода системы относится к классу задач модельно-ориентированной обратной разработки. В данной работе в качестве языка моде-

лирования используется язык AADL (Architecture Analysis & Design Language). Для построения моделей используется функциональное ПО для авионики, которое соответствует спецификации ARINC 653[3].

Целью данной работы является разработка и реализация алгоритма построения архитектурных моделей в формате AADL на основе автоматического анализа исходного кода функционального ПО на языке C для ARINC 653 совместимых ОС. В работе рассматривается двухэтапный алгоритм построения архитектурной модели функционального ПО. На первом этапе производится анализ исходного кода ПО и определяются ARINC 653 сущности и их связи. На втором этапе ARINC 653 сущности трансформируются в AADL модель. Предложенный алгоритм реализован на базе инструмента статического анализа CPAchecker [4].

## 2 Анализ исходного кода

Исследуемое функциональное ПО предоставляется в виде файлов с исходными кодами на языке C, в которых используются вызовы функций ОС, определенные спецификацией ARINC 653. Для демонстрации особенностей работы алгоритма используется пример программы, в которой создаются три процесса и четыре порта. Каждый из процессов реализует свою логику работы с портами. В листинге 1 представлен фрагмент ПО, в котором создается один процесс.

```
  static PROCESS_ID_TYPE pid_P1, pid_P2, pid_P3;
...
PROCESS_ATTRIBUTE_TYPE P1_process_attrs = {
        .PERIOD = 63000000LL,
        .TIME_CAPACITY = 63000000LL,
        .STACK_SIZE = 8096,
        .BASE_PRIORITY = 1,
        .DEADLINE = SOFT,
};
P1_process_attrs.ENTRY_POINT = shared_process2;
strncpy(P1_process_attrs.NAME, "P1_process",
sizeof(PROCESS_NAME_TYPE));
CREATE_PROCESS(&P1_process_attrs,&pid_P1,&ret_process);
```

**Листинг 1.** Создание ARINC 653 процесса.

Создание процесса происходит с помощью вызова функции CREATE_PROCESS, которой в качестве первого аргумента передается структура, содержащая атрибуты создаваемого процесса. В листинге 1 для задания атрибутов процесса используется переменная P1_process_attrs с типом PROCESS_ATTRIBUTE_TYPE. Поле ENTRY_POINT содержит указатель на потоковую функцию, т.е. функцию, которой передается управление при старте процесса. В рассматриваемом примере программы явный вызов потоковой функции shared_process2 отсутствует, при этом поле ENTRY_POINT содержит указатель на потоковую функцию, что позволяет ОС неявно вызывать потоко-

вую функцию при выполнении программы. Для проведения анализа потоковой функции в рамках рассматриваемого алгоритма вызов потоковой функции искусственно добавляется уже на этапе анализа программы.

Перед началом анализа программы CPAchecker преобразует исходный код анализируемой программы в автомат потока управления — Control-Flow Automaton (CFA). CFA представляет собой направленный граф, узлы которого соответствуют точкам в программе, а ребра — операциям. Анализа исходного кода программы выполняется за два прохода. При первом проходе определяются все создаваемые процессы и их атрибуты. При втором проходе происходит искусственное добавление вызовов всех используемых потоковых функций непосредственно в CFA перед последней программной инструкцией функции main. Для определения фактических значений переменных в любой точке программы используется алгоритм анализа явных значений (explicit-value analysis) [5].

При анализе функционального ПО для ARINC 653 совместимых ОС особый интерес представляет случай, когда одна и та же потоковая функция используется несколькими процессами, но с разными параметрами. В листинге 2 представлен фрагмент программы в котором функции first_process, second_process и third_process являются потоковыми функциями для трех различных процессов. Каждая из этих функций в свою очередь вызывает одну и ту же функцию shared_process, но с различными значениями для параметров read и write. Фактически функции first_process, second_process и third_process являются лишь оболочками для вызова потоковой функции shared_process с различными параметрами. В зависимости от значений фактических параметров при вызове функции shared_process в ней вызываются соответствующие функции для чтения и записи в порт READ_FROM_SAMPLING_PORT и WRITE_TO_SAMPLING_PORT. Для определения значения фактических параметров read и write используется алгоритм анализа явных значений. В результате для разных процессов внутри функции shared_process будут выполняться разные фрагменты кода.

```
static void first_process(void) {
  shared_process(true, false);
}
static void second_process(void) {
  shared_process(true, true);
}
static void third_process(void) {
  shared_process(false, true);
}
static void shared_process(bool read, bool write) {
  if (read)  {
    READ_FROM_SAMPLING_PORT(RT);
  }
  if (write) {
    WRITE_TO_SAMPLING_PORT(WAZ);
```

```
  }
}
```

**Листинг 2.** Использование параметров в потоковой функции.

При анализе потоковой функции, используемой несколькими процессами, может потребоваться информация о том, какому именно процессу она принадлежит. При создании процесса операционная система назначает ему идентификатор. В дальнейшем идентификатор текущего потока можно узнать с помощью вызова функции ОС GET_MY_ID. Учитывая, что при статическом анализе реальных вызовов функций ОС не происходит, на этапе анализа в программу вводится искусственная переменная $PID, в которой хранится идентификатор текущего процесса.

```
void CREATE_PROCESS (
  PROCESS_ATTRIBUTE_TYPE  *attributes,
  PROCESS_ID_TYPE         *process_id,
  RETURN_CODE_TYPE        *return_code)
```

**Листинг 3.** Прототип функции создания ARINC 653 процесса.

Для того чтобы определить какому именно процессу принадлежит потоковая функция при ее вызове в программе в CFA непосредственно перед ее вызовом добавляется операция присваивания переменной $PID значения идентификатора текущего процесса. На рисунке 1 представлен фрагмент CFA в графическом формате.

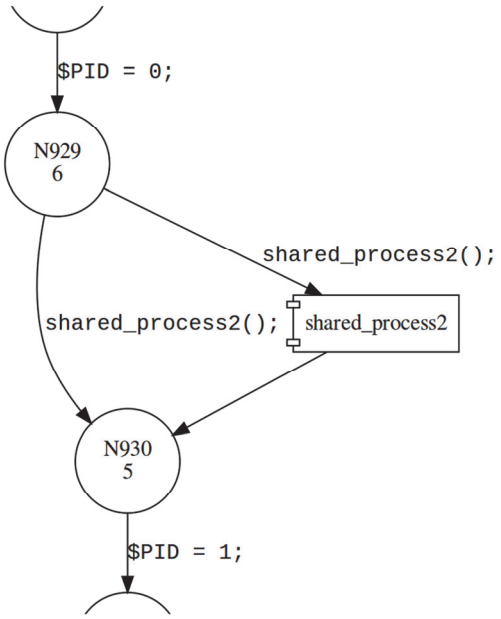


**Рис. 1.** Фрагмент автомата потока управления (CFA) программы.

```
static void shared_process2(void) {
  RETURN_CODE_TYPE ReturnCode;
  PROCESS_ID_TYPE MyId;
  GET_MY_ID ( &MyId, &ReturnCode );
               if (MyId == pid_P1) {
    READ_FROM_SAMPLING_PORT(RT);
    WRITE_TO_SAMPLING_PORT(WAZ);
  }
               if (MyId == pid_P2) {
    …
```

**Листинг 4.** Потоковая функция shared_process2.

Для рассматриваемого примера, в котором используется потоковая функции shared_process2, с помощью предложенного алгоритма анализа было определено, что:

1. в программе создаются три ARINC 653 потока и четыре порта;
2. первый поток обращается к порту RT на чтение и к порту WAZ на запись;
3. второй поток обращается к порту RDELTAE на чтение и к порту WQ на запись;
4. третий поток обращается к портам RT и RDELTAE на чтение и к портам WAZ и WQ на запись.

Информация, полученная в результате анализа исходного кода, сохраняется в программе во внутреннем формате и используется в дальнейшем при построении архитектурной модели. В следующем разделе будет описан второй этап алгоритма — этап построения архитектурной модели.

## 3 Построение архитектурной модели

На втором этапе на основании информации об имеющихся ARINC 653 объектах и логических связей между ними строится AADL модель. ARINC 653 сущности преобразуются в элементы AADL модели по определенным правилам: ARINC 653 раздел соответствует AADL процессу, ARINC 653 процесс соответствует AADL потоку, ARINC 653 порты без очереди (SAMPLING PORT) отображаются в AADL модели с помощью компонента data порт, порты с очередью (QUEUING PORT) отображаются в AADL модели с помощью компонента event data порт.

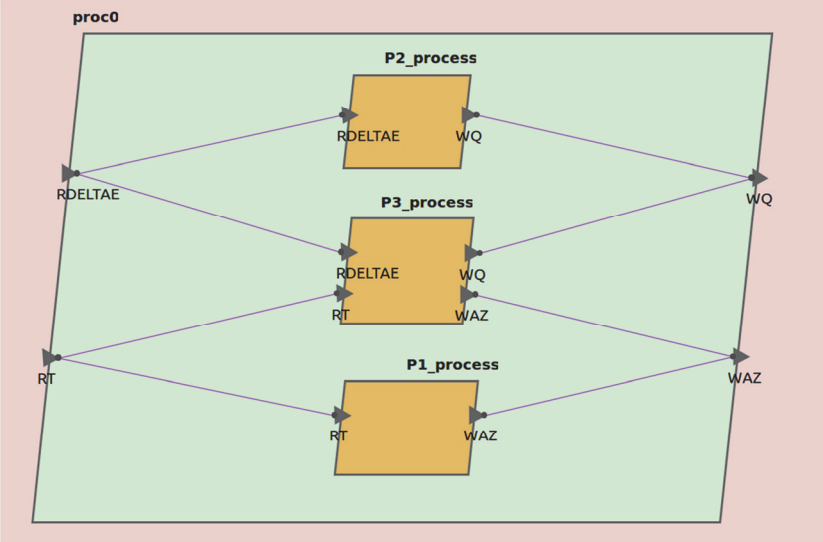


**Рис. 2.** Графическое представление AADL модели.

```
thread P1_process
features
  RT: in data port {Source_Name => RT;
    ARINC653::Sampling_Refresh_Period => 100 ms;};
  WAZ: out data port {Source_Name => WAZ;
    ARINC653::Sampling_Refresh_Period => 100 ms;};
properties
```

```
  Source_Stack_Size => 8096 Bytes;
  Priority => 1;
  Period => 63 ms;
  Compute_Deadline => 63 ms;
end P1_process;
```

**Листинг 5.** AADL поток.

В листинге 5 представлен фрагмент AADL модели в текстовом формате, содержащий описание AADL потока, который соответствует ARINC 653 процессу из листинга 1. В листинге 6 представлен фрагмент AADL модели содержащий реализацию AADL процесса. AADL процесс состоит из трех AADL потоков, которые соответствуют трем ARINC 653 процессам, использующим одну и ту же потоковую функцию shared_process2 из листинга 4. Эта потоковая функция устанавливает разные правила работы с портами для разных ARINC 653 процессов.

```
process implementation main_process.impl
subcomponents
  P1_process: thread P1_process.impl;
  P2_process: thread P2_process.impl;
  P3_process: thread P3_process.impl;

connections
  con0: port RT -> P1_process.RT;
  con1: port P1_process.WAZ -> WAZ;
  con2: port RDELTAE -> P2_process.RDELTAE;
  …
```

**Листинг 6.** Фрагмент AADL процесса.

Полученная архитектурная модель может использоваться для дальнейшего анализа и проектирования системы с помощью любого инструмента, поддерживающего спецификацию AADL версии 2.1. В данной работе был использован инструмент MASIW [6], который предназначен для проектирования и анализа систем интегрированной модульной авионики. На рисунке 2 представлено графическое представление AADL модели полученное с помощью инструмента MASIW.

## 4 Оценка производительности

Для оценки производительности алгоритма необходимо исследовать время выполнения этапа анализа исходного кода, на который приходятся основные временные затраты работы алгоритма. На рисунке 3 представлено три графика, отражающие зависимость среднего времени анализа исходного кода от сложности

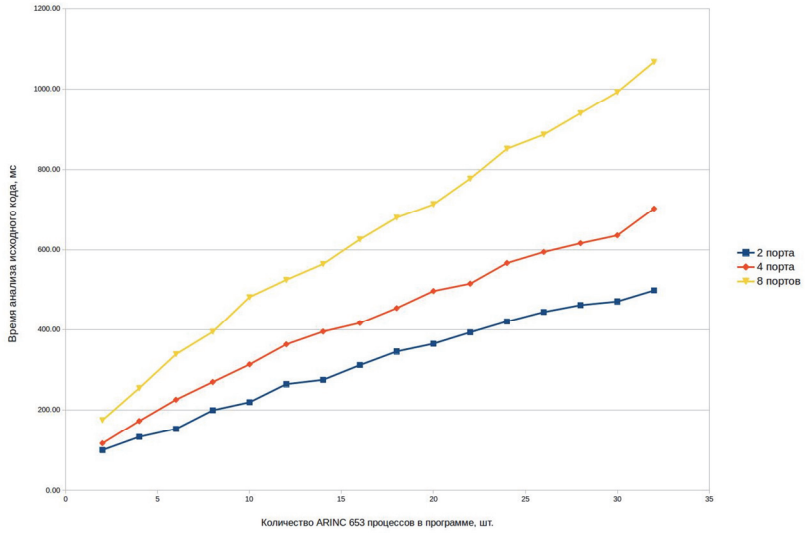


**Рис. 3.** График зависимости времени анализа исходного кода от сложности ПО.

анализируемого функционального ПО. Сложность ПО определяется количеством создаваемых ARINC 653 процессов и количеством используемых ресурсов (портов) в потоковой функции каждого процесса. Для исследования использовалось функциональное ПО, в котором количество ARINC 653 процессов последовательно увеличивалось с двух до тридцати двух с шагом два. Кроме этого, для каждого варианта конфигурации ПО, в котором количество процессов зафиксировано, использовались три разные потоковые функции: первая потоковая функция обращается к двум портам (нижний график), вторая – к четырем (средний график) и третья – к восьми (верхний график). Из графика видно, что при использовании самой простой конфигурации ПО среднее время выполнения этапа анализа исходного кода составляет около 100 миллисекунд. При использовании самой сложной конфигурации ПО среднее время выполнения этапа анализа исходного кода составляет немного больше одной секунды (1067 миллисекунд). Для всех остальных промежуточных конфигураций ПО значение среднего времени анализа не выходило за пределы диапазона значений от 100 до 1067 миллисекунд. Измерения проводились на компьютере с операционной системой GNU/Linux (Ubuntu 16.04) x86_64 с

процессором Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz и оперативной памятью 16 Гбайт.

## 5 Заключение

В данной работе представлен двухэтапный алгоритм построения архитектурных моделей в формате AADL на основе автоматического анализа исходного кода функционального ПО для ARINC 653 совместимых ОС. Предложенный алгоритм реализован в виде отдельного модуля для статического анализатора CPAchecker. Работа алгоритма продемонстрирована на тестовом приложении для ARINC 653 совместимой операционной системы.

## Литература

1. SAE International standard AS5506C, Architecture Analysis & Design Language (AADL), 2017. http://standards.sae.org/as5506c/.
2. SAE International standard AS5506/1A, Architecture Analysis & Design Language (AADL), Annex A: ARINC653 Annex, 2015. http://standards.sae.org/as5506/1a/.
3. ARINC. ARINC Specification 653P1-3: Avionics Application Software Standard Interface Part 1 - Required Services. Aeronautical Radio INC, Maryland, USA, 2010.
4. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg, 2011.
5. Beyer D., Löwe S.: Explicit-value analysis based on CEGAR and interpolation. Technical Report MIP-1205, University of Passau / ArXiv 1212.6542, December 2012.
6. Буздалов Д.В., Зеленов С.В., Корныхин Е.В., Петренко А.К., Страх А.В.,Угненко А.А., Хорошилов А.В. Инструментальные средства проектирования систем интегрированной модульной авионики. Труды Института системного программирования РАН. Том 26, выпуск 1, 2014.