DOI: 10.15514/ISPRAS-2021-33(6)-3

Модель и декларативный язык спецификации бинарных форматов данных

[1] *А.А. Евгин, ORCID: 0000-0001-8826-6898 <evgin@ispras.ru>*
[1,2] *М.А. Соловьев, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>*
[1,2] *В.А. Падарян, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] *Московский государственный университет имени М.В. Ломоносова,*
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Ряд задач, связанных с бинарными форматами данных, включает в себя задачи разбора, генерации и совместного анализа кода и данных. Ключевым элементом для решения всех этих задач является универсальная модель формата. В данной работе предлагается подход к моделированию бинарных форматов. Описанная модель обладает достаточной выразительностью для спецификации большинства распространенных форматов. Отличительной особенностью модели является гибкость при задании положений полей, а также возможность описывать внешние поля, структура которых не определяется при разборе. В реализованной инфраструктуре имеется возможность создания и модификации представления с помощью программных интерфейсов. Предлагается алгоритм для разбора данных по модели, основаный на понятии вычислимости полей. В работе также представлен предметно-ориентированный язык спецификации форматов. Указываются описанные форматы и потенциальные практические применения модели для программного анализа форматированных данных.

Ключевые слова: бинарные форматы данных; декларативное описание; анализ бинарного кода; совместный анализ кода и данных

Для цитирования: Евгин А.А., Соловьев М.А., Падарян В.А. Модель и декларативный язык спецификации бинарных форматов данных. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 27-50. DOI: 10.15514/ISPRAS–2021–33(6)–3

Model and declarative specification language of binary data formats

[1] *A.A. Evgin, ORCID: 0000-0001-8826-6898 <evgin@ispras.ru>*
[1,2] *M.A. Solovev, ORCID: 0000-0002-0530-6442 <icee@ispras.ru>*
[1,2] *V.A. Padaryan, ORCID: 0000-0001-7962-9677 <vartan@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia

Abstract. A number of tasks related to binary data formats include the tasks of parsing, generating and conjoint code and data analysis. A key element for all of these tasks is a universal data format model. This paper proposes an approach to modeling binary data formats. The described model is expressive enough to specify the most common data formats. The distinctive feature of the model its flexibility in specifying field locations, as well as the ability to describe external fields, which do not resolve into detailed structure during parsing. Implemented infrastructure allows to create and modify a model using application programming interfaces. An

algorithm is proposed for parsing binary data by a model, based on the concept of computability of fields. The paper also presents a domain-specific language for data format specification. The specified formats and potential applications of the model for programmatic analysis of formatted data are indicated.

Keywords: binary data formats; declarative formats specification; binary code analysis; conjoint code and data analysis

For citation: Evgin A.A., Solovev M.A., Padaryan V.A. Model and declarative specification language of binary data formats. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 6, 2021, pp. 27-50 (in Russian). DOI: 10.15514/ISPRAS–2020–33(6)–3

1. Введение

Необходимость обработки сложных структур данных возникает во всех сферах системного и прикладного программирования. Форматы, используемые для хранения и передачи данных, обычно описываются человекочитаемыми спецификациями, которые могут быть как общедоступными, так и закрытыми. Говоря о форматах данных, мы обычно подразумеваем наличие некоторого объекта со своими характеристиками, который необходимо представить в адресном пространстве (например, в памяти или в сетевом потоке). Форматированными данными мы называем данные, снабженные разметкой, содержащей информацию о структуре данных и сопоставляющей фрагменты данных с характеристиками объекта (полями формата).

Ряд задач, связанных с форматами данных, достаточно разнообразен и возглавляется основной задачей разбора данных. Для обработки данных определенного формата программисту необходимо реализовать средство разбора, которое будет восстанавливать структуру данных и характеристики представленного объекта. Подходы к решению этой задачи очень разнообразны и варьируются от ручной реализации программы разбора конкретного формата до реализации системы автоматической генерации разборщиков по спецификации.

С учетом быстрого роста количества разнообразных форматов, хорошо зарекомендовал себя именно второй подход. Были разработаны такие инструменты как DataScript [1], FlexT[2], Kaitai Struct [3]. Их основным достоинством является то, что они позволяют сжато описывать спецификации форматов и автоматически генерировать устойчивые к ошибкам разборщики. В сфере анализа сетевых прокотолов были разработаны инструменты, адаптированные к особенностям сетевых пакетов: PacketTypes [4], BinPAC [5], GAPAL [6]. При этом никакого принципиального ограничения на использование их для описания именно сетевых протоколов нет.

В основе всех этих инструментов лежит универсальная модель формата данных. Эта модель должна обладать достаточной выразительностью для описания всех встречающихся форматов данных. Важно то, что наличие такой модели позволяет не только разбирать данные произвольного формата, но и проводить совместный анализ кода и данных. К этому ряду задач можно отнести восстановление форматов, отслеживание форматированных данных при исполнении программы.

Помимо разбора и анализа данных, актуальной задачей является задача генерации данных заданного формата для фаззинга систем. Использование единой модели данных на всем цикле восстановление-генерация-разбор сильно упрощает работу исследователю.

Таким образом, ключевой задачей при работе с форматами является разработка модели формата. Помимо требований функциональной выразительности, к ней должны предъявляться требования, учитывающие особенности возникающих задач. Так, например, в некоторых задачах разбора возникает необходимость разбирать неполные данные. В задачах восстановления форматов актуальна возможность модифицировать представление формата. В системах обнаружения вторжений (IDS), работающих сразу с базой форматов, возникает необходимость автоматически распознавать формат по данным.

В вышеуказанных инструментах основной упор делается на функциональную выразительность и безопасность, при этом мало внимания уделяется универсальности модели. В данной работе представлена разработанная модель формата данных, отвечающая сформулированным требованиям, реализованная на базе системы анализа бинарного кода Glassfrog. В разд. 2 приводится описание конструкций, часто встречающихся в форматах данных, которые модель должна описывать. Разд. 3 посвящен обзору существующих подходов и доступных программных инструментов для работы со структурированными данными. В разд. 4 предлагается универсальная модель формата данных, описываются декларативный язык для спецификации форматов и алгоритм разбора данных на основе предложенной модели. Разд. 5 и 6 содержат описание тестирования разработанной системы и направлений дальнейшей работы. В заключении формулируются отличительные черты предложенной модели и предполагаемые практические приложения.

2. Форматы данных

Среди распространенных общедоступных форматов можно выделить несколько групп:

- форматы сетевых протоколов: TCP, DNS, IPv4, ICMP;
- форматы исполняемых файлов: ELF, DOS MZ, Microsoft PE;
- мультимедийные форматы: GIF, PNG, PSD;
- и т. д.

Ниже приведены часто встречающиеся в описаниях форматов конструкции. Источниками описаний являлись официальные спецификации (в т.ч. соответствующие документы RFC).

- **Переиспользование форматов.** Поля формата сами по себе имеют некоторую структуру. Некоторые поля формата могут иметь одну и ту же структуру. В таком случае естественным решением будет описать эту структуру отдельно и указать ее как формат всех этих полей.
- **Параметризация формата.** Некоторые форматы имеют версии, отличающиеся только размером некоторого поля. В таких случаях целесообразно воспринимать этот формат как универсальный, зависящий от параметра – размера поля. Аналогично форматы могут отличаться лишь порядком байтов в многобайтовых последовательностях. В таком случае также удобно использовать порядок байтов как параметр. Другим примером необходимости параметризации является инкапсуляция сетевых протоколов – во многих протоколах данные полезной нагрузки также имеют некоторый формат. Этот инкапсулированный формат можно рассматривать как параметр верхнего формата и таким образом реализовать инкапсуляцию. Например, описав отдельно форматы сообщений протоколов TCP и UDP, мы можем передавать их в качестве параметра в протокол IP и получать связки TCP/IP и UDP/IP.
- **Условная структура.** Простейший случай – это опциональные поля, которые могут присутствовать, а могут отсутствовать в зависимости от некого условия (значения другого поля). Например, в формате сжатого файла GZIP может храниться имя исходного файла или не храниться, в зависимости от значения флага FNAME в заголовке. В более сложном случае вся структура полей формата может зависеть от некоторого условия. Например, формат одной секции (chunk) в файлах изображения PNG задается полем типа этой секции и может состоять из полей метаинформации об изображении, сжатого содержимого изображения, данных о палитре или быть вообще пустым.
- **Адресация полей: точка отсчета и отступ.** Некоторые поля в соответствии со спецификацией располагаются на определенном отступе от предыдущего поля или от некоторой другой точки отсчета (например, начала сообщения), причем величина отступа может определяться динамически. Например, в формате исполняемого файла ELF содержатся таблицы заголовков программы и секций, отступ которых задается соответствующими полями в заголовке ELF.
- **Последовательности полей.** Часто встречаются форматы, в которых поля одной структуры повторяются некоторое количество раз. Это количество может быть фиксированным или определяться значениями других полей. Кроме того, последовательность полей может быть терминированной. Например, в формате сообщения протокола HTTP при использовании опции «Transfer-Encoding: chunked» структуры Chunk («куски») повторяются до тех пор, пока не встретится т.н. «нулевой кусок» (Chunk с телом нулевой длины). Другой пример: в формате сообщения протокола DNS поле заголовка QDCOUNT задает количество структур запросов в секции запросов.
- **Битовые поля.** В некоторых форматах размеры полей не выровнены по границам байтов. Типичным примером являются флаги – поля, состоящие из одного бита.
- **Потоковые данные.** Некоторые поля содержат сжатые или зашифрованные данные. В таком случае значение слова декодируется как битовый поток и может иметь переменную длину, определяемую динамически.
- **Валидация значений.** Поля форматов могут иметь фиксированное значение, определенное его спецификацией (т.н. magic-поля). В ряде случаев в спецификации указывается, что в случае несоответствия этому значению дальнейший разбор проводить не следует. Другим примером валидации значений являются поля контрольных сумм, используемые, например, в сетевых протоколах TCP и UDP.
- **Управление порядком байтов.** В некоторых форматах порядок байтов фиксируется в спецификации, однако есть форматы, которые хранят в себе данные переменного порядка байтов, причем определяется он динамически. Примером является формат исполняемого файла ELF.

Надо отметить, что в данной работе помимо достаточно сложных форматов (ELF, DNS, и т.п.) рассматриваются и более простые. Так, такие виды структурированных данных как структуры языка Си, списки, целые числа, также рассматриваются как форматы. Часто они используются как базовые форматы для построения более сложных форматов. В контексте задач анализа они также являются полезными для исследователя, поскольку позволяют типизировать тривиальные данные.

3. Подходы к описанию форматов

3.1 Синтаксический анализ

Понятие формата данных тесно связано с понятием формального языка. Классическая теория формальных языков описывается, например, в [7]. Формальный язык определяется следующим образом: пусть дано некое непустое конечное множество, называемое алфавитом, тогда словом называется произвольная (возможно, пустая) конечная последовательность элементов алфавита (символов), а языком – произвольное множество слов. Формат данных мы можем представить, как формальный язык. Например, текстовый формат сетевого протокола HTTP определяет способ записи сообщения в виде конечной последовательности символов ASCII, что, соответственно, задает некоторое множество слов в алфавите ASCII, т.е. некий язык. Аналогично, бинарный формат данных, например, формат исполняемого файла ELF, хранящийся в памяти, можно представить, как язык над алфавитом байтов.

Задача задания формального языка (в нашем контексте – описание формата) является базовой в теории формальных языков и имеет набор стандартных решений. Одним из базовых представлений языка являются формальные грамматики, классифицируемые с помощью иерархии Хомского по степени жесткости условий. Наиболее ограниченным типом

грамматик задаются регулярные языки, которые описываются регулярными выражениями. В рамках разработки языков программирования для синтаксического анализа наибольшее распространение получили контекстно-свободные грамматики (КС-грамматики).

Синтаксический анализ восстанавливает структуру разбираемого текста в соответствии с заданным языком. Этот вид анализа достаточно близок к задаче разбора данных в соответствии с заданным форматом. В теории формальных языков описываются алгоритмы построения машины разбора (анализатора) по КС-грамматике: LR-, LL(k)-, LALR-анализаторы и т.д., которые имеют доказательную базу своих свойств и хорошо показали себя за многолетнюю практику использования. Для записи КС-грамматик в компьютерной сфере часто используют форму Бэкуса-Наура (БНФ) или ее расширенную модификацию РБНФ.

Существуют инструменты, генерирующие разборщики по подобному представлению грамматики:

- yacc – программа для генерации разборщика по грамматике, описанной в форме БНФ с небольшими модификациями (вставками на языке Си), генерирует LALR-анализатор в виде кода на языке Си;
- Bison – GNU-версия yacc, которая может генерировать также LR-, IELR(1)- и GLR-анализаторы;
- ANTLR – использует для описания грамматики форму РБНФ и генерирует LL(*)-анализатор на различных выходных языках (Java, C#, C++, JavaScript, Python, Swift, Go).

Bison и yacc требуют для своей работы предварительной токенизации входных данных, т.е. лексический анализ (используются инструменты Lex или Flex).

3.2 Грамматики с зависимостями по данным

Синтаксический анализ имеет принципиальную особенность, не позволяющую использовать его для разбора форматов – независимость синтаксической структуры от *значений* ее элементов. Когда речь идет о произвольных форматах данных, структура результата разбора может меняться в зависимости именно от значений своих элементов. В таком случае поле разбираемой структуры интерпретируется, например, как число или строка, и используется в некотором выражении, определяющем структуру: условии, длине и т.п. Такую задачу называют «разбор с зависимостями по данным» (data-dependent parsing) и именно она соответствует задаче разбора форматов файлов и сетевых пакетов. Примером может являться распространенный шаблон формата: «поле длины + тело», где в начале задается фиксированного размера слово, интерпретируемое затем, как число, определяющее количество символов в «теле». Подобные форматы соответствуют формальным языкам, однако их описание с помощью классических контекстно-свободных грамматик затруднительно.

Формальные грамматики получили огромное функциональное развитие: атрибутные грамматики [8], грамматики, разбирающие выражения (PEG) [9] и т.д., что существенно повысило их выразительную мощность. Генерируемые на их основе разборщики являются модификациями вышеупомянутых «канонических» синтаксических анализаторов. Наиболее близкими к задаче разбора бинарных форматов можно считать т.н. грамматики с зависимостями по данным (data-dependent grammars, ЗД-грамматки). ЗД-грамматики являются расширением КС-грамматик и дополняют их привязанными семантическими значениями, которые могут быть использованы для наложения условий и вычисления выражений в процессе разбора. На их основе были разработаны такие инструменты для разбора форматов по описанию, как Yakker [10] и Iguana [11].

В работе [10] в качестве модели разборщика ЗД-грамматик описывается автомат с зависимостями по данным (data-dependent automaton), а также формально доказывается, что его мощности достаточно для разбора языков, задаваемых ЗД-грамматиками. Предлагаемый

авторами алгоритм разбора является модификацией хорошо известного алгоритма Эрли [12]. При этом авторы ставят перед собой задачу максимально эффективно использовать уже существующие алгоритмы разбора и их реализации, поскольку они за долгое время использования были хорошо оптимизированы. В связи с этим для спецификации формата было предложено использовать дополнительный язык промежуточного уровня, максимально близкий к языку описания контекстно-свободных грамматик [13].

Несмотря на достаточную выразительность моделей для описания практически произвольных форматов данных, в приведенных работах основной упор делается все ещё на разбор языков программирования. Подход с использованием модификаций грамматик все ещё страдает от недостаточного удобства для описания именно бинарных форматов: отсутствует возможность лаконично описывать такие опции, как порядок байтов и битовые поля. Тем не менее очевидным достоинством такого подхода является строгая доказательная база и высокая эффективность разборщиков, что является важной особенностью, когда речь идет о разборе языков программирования общего назначения.

В работе [14], однако, высказывается мнение, что подход с использованием КС-грамматик может быть эффективен в случае построения *валидатора* входных данных, который часто является частью разборщика. Авторы изучили конструкции сетевых протоколов с формальной точки зрения и предложили систему языков с доказательной базой эффективности их разбора. В частности, такие конструкции, как поля длины, проверки равенства, неравенства и выполнения численного сравнения, были описаны ими в формальном стиле.

3.3 Предметно-ориентированные языки

Альтернативный подход к описанию форматов данных заключается в разработке собственной модели формата и предметно-ориентированного языка, который будет транслироваться в нее. Такой подход позволяет учесть все требования к выразительности описания, сформулированные выше, и поэтому является более предпочтительным. При этом синтаксис и семантика такого языка будут закономерно сложнее тех, что использовались при описании грамматик.

Синтаксический вид языка может выбираться разработчиком исходя из соображений максимального удобства описания спецификаций. В таком случае разработчики часто останавливаются на конструкциях, сходных с описанием структур Си. На рис. 1 приводится пример описания формата ELF на языке спецификации форматов DataScript [1].

Другие авторы отталкивались от удобства разработки и избегали использования собственного синтаксиса. В таком случае за основу часто брался язык разметки (XML, YAML и т.п.) с определенной схемой, и разбор синтаксиса перекладывался на стандартный инструмент разбора данного языка. Так, например, в инструменте Kaitai Struct [3] для описания форматов используется язык YAML (рис. 2), а в системе фаззинга Peach Fuzzer [15] – язык XML.

Альтернативой использования декларативного языка описания спецификаций является использование API для конструирования представления формата. Такой подход используется, например, в инструменте обратной разработки и фаззинга Netzob [16] для описания форматов сообщений. Их декларативный язык ZDL является открытым API библиотеки Netzob, а файлы спецификаций представляют собой файлы кода на языке Python (рис. 3).

```
ElfSection(Elf32_File.Elf32_SectionHeader h) {
  Elf32_File::h.sh_offset:
    union {
        { } null : h.sh_type == SHT_NULL;
        StringTable(h) strtab : h.sh_type == SHT_STRTAB;
        SymbolTable(h) symtab : h.sh_type == SHT_SYMTAB;
        SymbolTable(h) dynsym : h.sh_type == SHT_DYNSYM;
        RelocationTable(h) rel : h.sh_type == SHT_REL;
        ...
    } section;
};

SymbolTable(Elf32_File.Elf32_SectionHeader h) {
    Elf32_Sym {
        uint32    st_name;
        uint32    st_value;
        uint32    st_size;
        uint8     st_info;
        uint8     st_other;
        uint16    st_shndx;
    } entry[h.sh_size / sizeof Elf32_Sym];
};
```

Рис. 1. Фрагмент описания формата ELF на языке DataScript
Fig. 1. ELF format specification in DataScript

```
meta:
  id: tcp_segment
  endian: be
seq:
  - id: src_port
    type: u2
  - id: dst_port
    type: u2
  - id: seq_num
    type: u4
  - id: ack_num
    type: u4
```

Рис. 2. Фрагмент описания сообщения протокола TCP на языке Kaitai Struct
Fig. 2. TCP message specification in Kaitai Struct

```
from netzob.all import *
f1 = Field(String(), name="field1")
f2 = Field(Integer(interval=(10, 100)), name="field2")
f3 = Field(Raw(nbBytes=14), name="field3")
symbol = Symbol([f1, f2, f3], name="symbol_name")
```

Рис. 3. Фрагмент описания на языке Netzob Description Language (ZDL)
Fig. 3. Modeling in Netzob Description Language (ZDL)

Такой императивный подход упрощает реализацию инструмента, поскольку авторам не требуется разрабатывать транслятор, однако синтаксические структуры получаются достаточно объемными, что увеличивает размер спецификации.

В синтаксическом анализе математическими объектами, на которых строится спецификация формата, являются *грамматика* и *формальный язык*. В основе моделей данного подхода часто лежат такие математические объекты, как *ориентированный граф* и *дерево*.

Вершинами первого обычно являются форматы данных, а ребра графа отражают появление указанного формата в описании полей исходного. Для корректной спецификации формата при этом необходимо определить расположение полей, например, пронумеровав ребра. Древовидные структуры часто возникают тогда, когда используются встроенные в модель форматы, описание которых не требуется. На рис. 4 приводится схематичное изображение модели в DataScript. На основе данной модели строится конкретный экземпляр спецификации, корневым элементом которого является глобальный *StructType*.

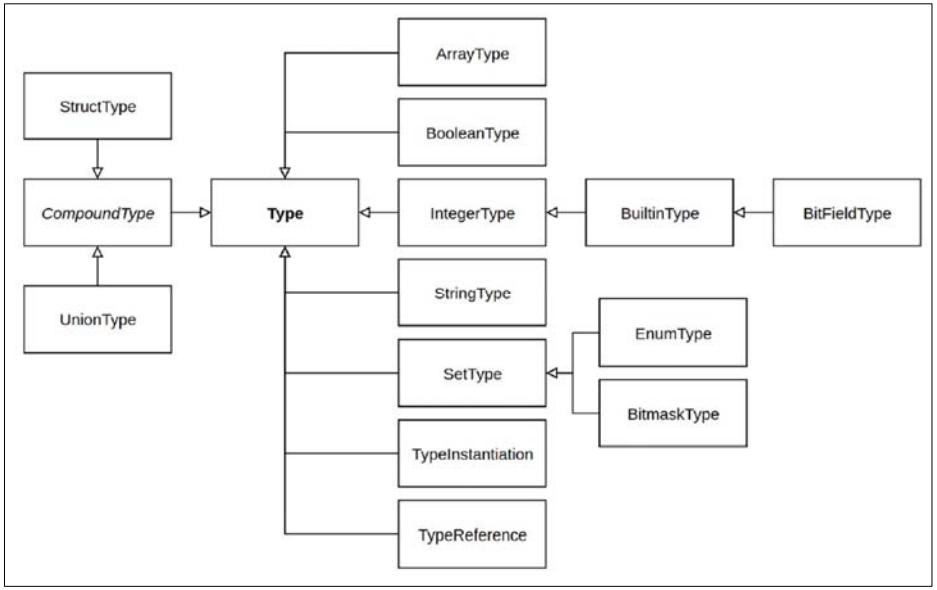

Рис. 4. Схематичное изображение модели в DataScript
Fig. 4. Types model schema in DataScript

Древовидные структуры в моделях позволяют описывать форматы с контекстом. Иными словами, появляется возможность использовать поля вышеопределенного формата. В случае отсутствия контекста, в модели может быть предусмотрено использование параметров формата для аналогичных целей.

Листовыми вершинами в графах форматов часто являются встроенные форматы. Наиболее распространенные встроенные форматы отражают основные типы переменных в языках программирования:

- байтовый массив;
- целые числа различных размеров;
- булево значение;
- битовый массив;
- строка (ASCII или Unicode).

Модели, нацеленные на описания специфичных форматов данных, например, сетевых протоколов, могут также иметь специфические встроенные форматы, такие как IP-адреса, битовые маски, строковые разделители (CRLF). Для построения пользовательского формата из встроенных форматов используются агрегирующие форматы, такие как структура или массив.

Помимо этого, модели часто реализуют специальные директивы для описания сжатых полей (например, с помощью алгоритма GZIP). Такие директивы часто реализованы вручную заранее описанным алгоритмом декодирования. Некоторые модели относят кодирование строк к этому же случаю и позволяют описывать строки различных кодировок.

3.4 Существующие инструменты

Инструменты для описания форматов данных можно разделить по сферам применения. Исторически одной из первых постановок задачи декларативного описания формата была постановка в контексте описания сообщений сетевых протоколов, появившаяся в период активного развития сети Интернет. Тем не менее многие из этих инструментов не имеют принципиальных ограничений на использование для описания произвольных форматов данных. Авторы работы [4] представили свой инструмент PacketTypes как язык спецификации сетевых пакетов, способный описывать конструкции, встречающиеся в сетевых протоколах: например, инкапсуляция протоколов, поля переменного размера и опциональные поля. Авторы в том числе стремились с помощью автоматической генерации разборщиков решить проблему безопасности кода, работающего с сетями.

В дальнейшем инструменты для описания сообщений сетевых протоколов часто интегрировались в различные системы обнаружения вторжений (NIDS): например, инструмент BinPAC [5] был интегрирован в IDS Bro. Авторы BinPAC также уделяли большое внимание безопасности кода разборщика. Основной группой форматов, на которую они ориентировались, были сетевые протоколы уровня приложения. В их работе также отмечается, что для описания сообщений сетевых протоколов инструменты, основанные на грамматиках, такие как yacc и ANTLR недостаточно выразительны. Той же группой авторов в дальнейшем был разработан аналогичный инструмент Spicy [16], который также интегрирован в IDS Zeek (последователь Bro), однако позиционируется как генератор разборщиков произвольных форматов.

При разработке инструментов описания сетевых протоколов перед разработчиками встают задачи описания не только форматов сообщений, но также автоматов состояний протоколов и сессий. В связи с этим в подобных инструментах часто для разбора данных и поддержания состояний других элементов используется *внутренний движок*. В работе [6] описывается система GAPA (Generic Application-Level Protocol Analyzer) и используемый в ней язык GAPAL. GAPA ориентирован на анализ всего сетевого трафика на всех слоях. В GAPAL основным примитивом для описания является *протокол*, включающий в себя описания нижележащих протоколов, грамматики сообщений, автомата состояний и обработчиков сообщений. Для описания форматов сообщений GAPA использует комбинирование подхода описания Си-подобными структурами и КС-грамматиками. Как указывают авторы, их главной задачей было поддерживать описания бинарных и текстовых протоколов внутри одного представления.

Параллельно с инструментами анализа сетевого трафика та же задача декларативного описания формата ставилась и для произвольных форматов. В работе [19] описывается декларативный язык DataScript. Функциональной выразительности языка достаточно для описания очень широкого класса форматов. Однако гибкости в способе задания отступов не всегда хватает для описания пересекающихся или произвольно расположенных полей. Как отмечается автором, декларативное описание формата может быть использовано не только для разбора данных, но и для генерации данных описанного формата.

Как и в сфере сетевых протоколов, к инструментам разбора произвольных форматов предъявлялись высокие требования безопасности разборщиков. Так, в работе [18] авторами предлагается математический аппарат – исчисление описаний форматов (data description calculus, DDC) – основанный на теории типов, и на его основе формулируется ряд утверждений касательно разборщиков и возникающих в них ошибках. На основе этого аппарата авторы проанализировали языки инструментов PacketTypes и DataScript. В инструменте PADS [19], разработанном авторами вышеуказанного аппарата, особое внимание уделяется механизмам обработки ошибок в разборщиках.

В системах фаззинга выделяют два подхода к получению данных: мутация и генерация. Первый подход подразумевает внесение случайных изменений в существующие входные данные. Во втором подходе возникает необходимость описывать форматы данных. Подсистема фаззера принимает на вход описание формата и генерирует данные в соответствии с предоставленной спецификацией. В связи со спецификой использования представления формата, в инструментах обычно не реализована возможность разбирать данные. Часто встречающейся разновидностью фаззинга является фаззинг сетевых протоколов, по этой причине в данных инструментах для описания форматов можно встретить примитивы, характерные для сетевых протоколов (например, примитивы протокола HTTP).

Отличительной особенностью таких моделей является повышенное внимание к валидируемым полям: т.н. magic-полям, контрольным суммам и т.п. В моделях встречаются встроенная поддержка ограничений на значения полей и предикаты корректности.

Многие из данных инструментов (BooFuzz [20], Netzob [16]) не используют декларативный язык для описания форматов, а вместо этого позволяют создавать экземпляры форматов с помощью API. Однако один из наиболее известных инструментов фаззинга Peach Fuzzer [15] (в 2020 году был выкуплен компанией GitLab и распространяется под названием GitLab Protocol Fuzzer [21]) получает на вход описание модели данных в виде XML-файла определенной схемы.

Среди инструментов, способных обрабатывать форматированные данные, отмечаются также редакторы бинарных файлов. Изначально поддержка форматов обеспечивалась в них по первому сценарию разработки разборщика, т.е. реализовывалась вручную для каждого необходимого формата. Описание пользовательских форматов было весьма ограничено и сводилось к простым структурированным полям фиксированной длины. На текущий момент были разработаны такие инструменты как 010 Editor [22] и GNU poke [23], которые позволяют описывать пользовательские форматы произвольной сложности. Такие инструменты имеют графический интерфейс, что существенно повышает наглядность и позволяет визуализировать процесс разбора данных.

Для изучения свойств представления формата необходим доступ к исходному коду инструмента. Некоторые из упомянутых инструментов (GAPA, 010 Editor) не имеют открытого кода, поэтому их изучение затруднено.

В табл. 1 представлена краткая сводка по рассмотренным инструментам, способным разбирать бинарные форматы. Среди критериев сравнения выделяются в первую очередь интерфейсы взаимодействия с инструментом: входной язык описания формата и интерфейс разбора данных. Из таблицы можно заключить, что большинство инструментов используют язык с собственным синтаксисом. Многие из этих языков имеют внешнее сходство в синтаксисе со структурами в языке Си (DataScript, BinPAC, PADS, Nail), другие содержат синтаксические конструкции, схожие с правилами вывода в грамматиках (Spicy, GAPAL). Некоторые инструменты используют для декларативного описания синтаксис языков представления данных YAML или XML (NetPDL в системе NetBee, Kaitai Struct).

Среди изученных инструментов только Netzob и BooFuzz представляет возможность манипулировать представлением формата с помощью API, который при этом является и

единственным способом спецификации формата. Таким образом в большинстве случаев разработчики не предоставляют возможности создавать экземпляры представлений программным способом.

Табл. 1. Существующие инструменты работы с форматами данных
Table 1. Existing data formats tools

	Спецификация формата	Разборщик	Модель	API модели	Открытый код	Поддержка проекта
Kaitai Struct	YAML-based	кодогенерация (C++, Java, Python, ...)	дерево	×	✓	✓
FlexT	DSL[1]	внутренний интерпретатор	?	×	×	✓
DataScript	DSL	кодогенерация (Java)	оргграф с корнем	×	✓	×
PADS	DSL	кодогенерация (C, OCaml)	–	×	✓	×
Spicy	DSL	кодогенерация (C++)	оргграф	×	✓	✓
BinPAC	DSL	кодогенерация (C)	–	×	✓	×
NetBee	XML-based	внутренний	–	×	✓	×
GAPAL	DSL	внутренний	?	?	×	?
PacketTypes	DSL	кодогенерация (C)	?	?	×	?
GitLab Protocol Fuzzer (ex-Peach)	Peach Pit (XML)	–	оргграф	×	*Community Edition*	✓
BooFuzz	API	–	–	✓	✓	✓
Netzob	API	–	–	✓	✓	×
FormatFuzzer	DSL (010 Editor)	встроенный	дерево	×	✓	✓
010 Editor	DSL	встроенный	?	×	×	✓
GNU poke	DSL	встроенный	–	×	✓	✓

Другим критерием сравнения является способ дальнейшего взаимодействия с представлением формата. Можно условно выделить два сценария работы данных инструментов: внутренний и внешний. Внутренний сценарий подразумевает трансляцию описания формата в промежуточное представление (модель формата) и дальнейшее использование объекта этой модели модулями системы (например, для разбора данных). Внешний же сценарий может быть рассмотрен как дополнение к внутреннему: в нем на основе описания формата инструментом генерируется код разборщика формата на одном из поддерживаемых языков программирования. Этот процесс кодогенерации может быть

[1]DSL – domain-specific language (предметно-ориентированный язык)

рассмотрен как следующая фаза после трансляции описания формата в модель, т.е. внешний сценарий может дополнять внутренний. В табл. 1 в соответствующем столбце в скобках указываются целевые языки, на которых инструмент имеет возможность генерировать разборщики.

3.5 Выводы

Основываясь на изучении существующих инструментов и подходов к решению задачи декларативного описания формата данных, можно однозначно заключить, что не было найдено достаточно универсальной модели для покрытия заданного набора задач, связанных с форматированными данными. Многие инструменты качественно решают задачу генерации универсальных разборщиков, но не предоставляют возможности использовать модели формата для задач совместного анализа кода и данных. Несмотря на большую выразительность многих декларативных языков, большинство из них неявно используют предположение об определенном способе хранения данных (сетевой поток, файл и т.п.), что затрудняет реализацию таких опций, как разбор с частично недоступными данными.

4. Предлагаемое решение

Разрабатываемая в ИСП РАН система анализа бинарного кода *Glassfrog* [24] покрывает задачи совместного анализа кода и данных и поэтому нуждается в спецификации форматов данных. Ключевым элементом в системе *Glassfrog* является платформонезависимое представление кода *Pivot* [25]. Предлагаемая модель формата данных была реализована как часть этого представления и в связи с этим имеет доступ к другим его сущностям.

Система *Glassfrog* представляет возможность описывать архитектуры наборов команд (ISA), их синтаксис и семантику. Для этого разработан предметно-ориентированный язык *Ribbit*, который транслируется во внутреннее представление *Glassfrog* (*Pivot*-модуль); этот же язык используется для спецификации форматов данных. На рис. 5 представлен стандартный тракт обработки форматированных данных с помощью системы *Glassfrog*. Ниже в работе описываются разработанная модель формата и алгоритм разбора данных по ней, приводятся примеры спецификации форматов на языке *Ribbit*.

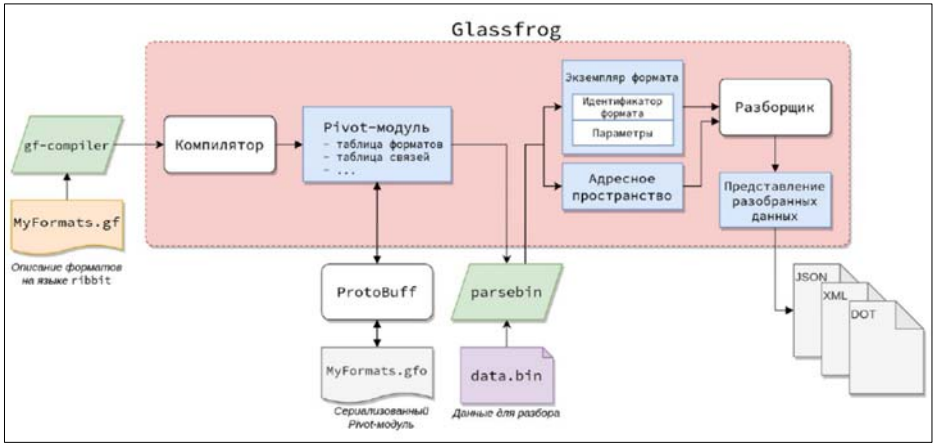

Рис. 5. Тракт разбора данных в системе Glassfrog
Fig. 5. Parse pipeline in Glassfrog

4.1 Модель формата

Ключевым элементом данной модели является формат (*DataFormat*). Каждый формат имеет набор параметров (*DataArg*), значения которых могут быть использованы при определении его структуры. Задав значения всех параметров определенного формата, мы таким образом конкретизируем его. Формат с выбранными значениями его параметров называется *экземпляром формата*. Сам по себе формат не является завершенной сущностью, разбор данных можно проводить только по экземпляру формата.

Основным отличием предлагаемого подхода от большинства изученных является отхождение от концепции полей как последовательных отрезков байтов в буфере. В модели предлагается трактовать поля как связи (*Relation*) между различными форматами. Каждый формат имеет набор (вообще говоря, неупорядоченных) связей. Все связи разделяются на три типа (рис. 6):

- внутренняя связь (*Internal*);
- внешняя связь (*External*);
- связь-значение (*Value*).

Внутренние связи описывают концепцию обыкновенных полей формата. В самом простом случае они состоят из экземпляра формата и структуры, описывающей ее положение. На этапе разбора данных для каждой внутренней связи рекурсивно разбирается ее формат данных.

Поскольку от модели требуется описание условных конструкций в структуре формата, указанный экземпляр формата и положение могут обособляться предикатами, определяющими выбор одного формата (или положения) из нескольких возможных. Итоговая структура представляет собой конструкцию, напоминающую switch-case: в представлении хранится массив из пар «предикат + вариант реализации». Вычисление предикатов осуществляется по порядку. Результатом вычисления такой конструкции будет тот вариант реализации, который соответствует первому найденному истинному предикату. Сам предикат представляет собой функцию, аргументами которой могут являться значения аргументов формата или значения других связей (внутренних или связей-значений).

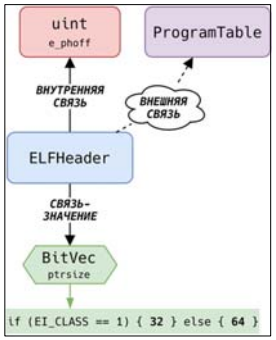

Рис. 6. Пример связей в заголовке ELF
Fig. 6. ELF header relations example

Внешние связи по своему строению не отличаются от внутренних: они также состоят из экземпляра формата и положения. Однако в отличие от внутренних, на этапе разбора данных форматы внешних связей не разбираются рекурсивно. В результате разбора будет храниться только структура экземпляра формата вместе с вычисленным положением. Это позволяет пользователю контролировать, какие связи он хочет разбирать, а какие хочет лишь указать. Также это позволяет разбирать неполные данные: разобрав имеющуюся часть, пользователь

отмечает поля из недоступной области как внешние и тем самым сохраняет информацию об их форматах, не разбирая их.

Связи-значения отличаются от предыдущих по своему строению. Они не имеют положения и представляют собой вычисляемое выражение некоторого типа. Значения этих связей могут быть использованы при определении формата или положения других связей. На этапе разбора данных эти значения вычисляются и сохраняются в результате разбора. Помимо удобства пользователя при определении других связей (сокращения количества кода), это позволяет описывать поля, состоящие из произвольного числа битов.

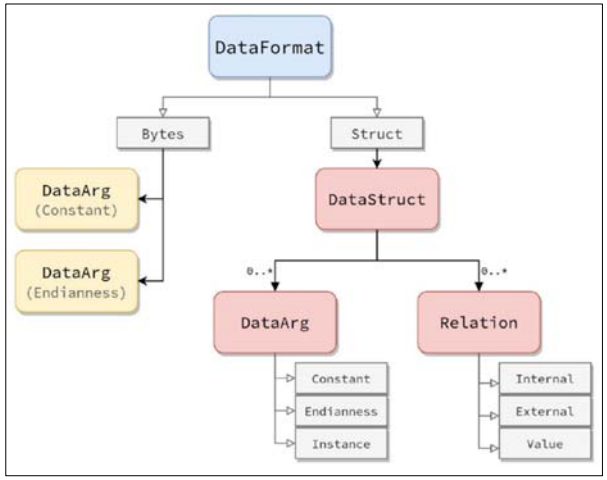

Рис. 7. Модель формата данных
Fig. 7. Data format model

В модели используется единственный встроенный формат *Bytes* – формат байтов определенного размера с определенным порядком байтов (little-endian/big-endian), используемым при чтении. Этот формат не имеет связей и по сути является самым примитивным способом структурировать данные. Все остальные форматы являются составными и определяются пользователем (*DataStruct*). На рис. 7 схематично иллюстрируются встроенный и пользовательские форматы.

Важным элементом модели является структура, описывающая положения внутренних и внешних связей (*Location*). Так же, как и формат связи, ее положение может обособляться предикатами, позволяющими описывать его как условное выражение. В результате поле формата может иметь динамически определяемое положение. Сама структура состоит из трех частей (рис. 7).

- **Объект отсчета** (*Origin*) – определяет, к какому объекту будет привязана связь. Возможны три способа привязки: к другой связи (*Relation*), к описываемой структуре (*Data*) и ко всему адресному пространству (*Const*). Привязка к другой связи определяется при помощи ее идентификатора (*RelationId*).
- **Якорь** (*Anchor*) – объект отсчета определяет связь, структуру или пространство, то есть во всех случаях некоторый диапазон адресов, от начала и до конца объекта. Якорь указывает, что именно необходимо выбрать: начало или конец объекта.
- **Отступ** (*Offset*) – задает отступ в байтах от определенной двумя предыдущими частями точки.

Стандартным положением связи считается конец предыдущей описанной связи с нулевым отступом. Такие положения автоматически выводятся компилятором *Glassfrog* и пользователю не придётся описывать их для каждого поля.

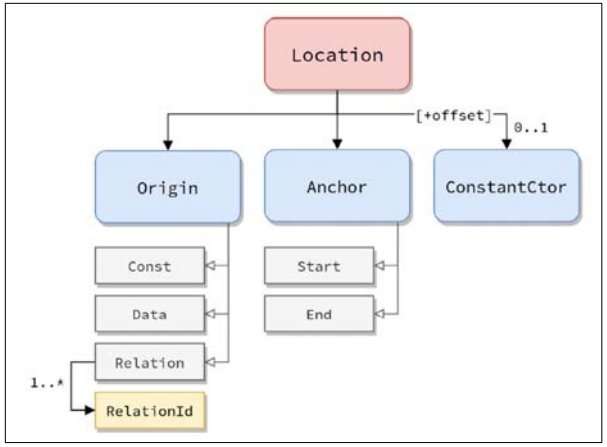

Рис. 8. Положение связи
Fig. 8. Relation location

Для представления форматированных данных используется структура *Data* и вспомогательная структура *DataPtr* (рис. 8). Структура *Data* хранит только разметку и не хранит сами данные внутренних связей. Для каждого вида связи хранятся соответствующие объекты:

- для *Internal* хранится объект *Data*;
- для *External* хранится объект *DataPtr*;
- для *Value* хранится значение одного из трех типов: битовый вектор, порядок байтов (*endianness*) или экземпляр формата.

Здесь видно, что для внешних связей не хранится внутреннее устройство их формата, а только его параметры и адрес, где находится связь.

На данном этапе реализации значения связей (внутренних и связей-значений) не валидируется. В дальнейшей работе планируется добавить обособление связей предикатами, вычисляемыми в процессе разбора и накладывающими ограничения на значения этих связей (аналог *assert*). В случае ложности данного предиката алгоритм разбора будет сразу завершаться, возвращая соответствующее исключение. Это позволит корректно обрабатывать такие примитивы как поля с фиксированным значением (magic-поля), поскольку в спецификациях подобных форматов часто указывается, что дальнейший разбор в случае несоответствия magic-поля требуемому значению проводить не следует.

Отличительной особенностью данной модели является исключительная гибкость в задании положений полей. Это позволяет описывать такие сложные структуры как закодированное доменное имя в сообщениях протокола DNS, не прибегая к написанию дополнительного кода. Интегрированность модели в представление *Pivot* даёт потенциал для использования модели в задачах совместного анализа кода и данных.

4.2 Алгоритм разбора

Поскольку предложенная модель существенно отличается от представления грамматик, для разбора не мог быть адаптирован один из существующих алгоритмов, и описываемый алгоритм был разработан «с нуля». На текущем этапе к алгоритму не предъявлялось

требований относительно скорости работы, однако такие требования, как завершимость, были удовлетворены (гарантируется, что разборщик не войдет в вечный цикл). Псевдокод алгоритма приводится в приложении 1.

Основным принципом работы алгоритма является принцип вычислимости. Считается, что связь вычислима, если вычислимы ее предикаты, положение и экземпляр формата (для внутренних и внешних связей) или определяющее ее выражение (для связей-значений). Положение связи может зависеть от других связей и станет вычислимым только тогда, когда все они будут вычислены. Аналогично параметры формата могут зависеть от других связей и экземпляр станет вычислимым, когда все они будут вычислены. Важно отметить, что в случае если предикат связи вычислим и истинен, другие предикаты и привязанные к ним экземпляры форматов вычисляться не будут. Когда экземпляр формата внутренней связи вычислен, алгоритм рекурсивно начинает вычислять его структуру.

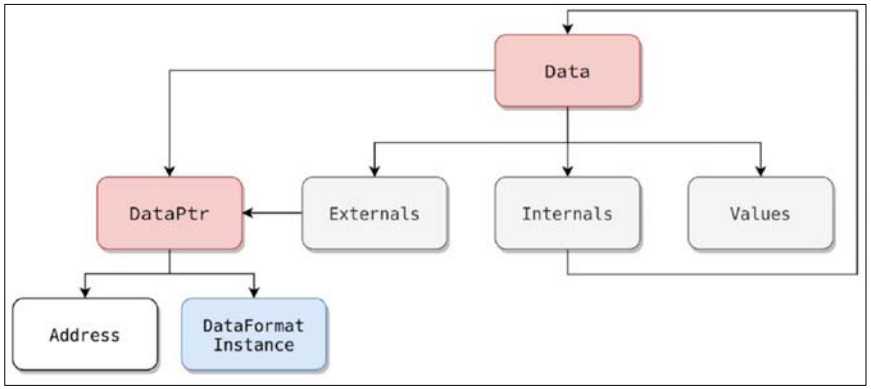

Рис. 9. Представление результата разбора
Fig. 9. Parse result representations

В процессе своей работы алгоритм определяет вычислимые на данном этапе связи и вычисляет их, сохраняя результат в структуру *Data* (рис. 9), и затем переходит на следующий этап. Если на данном этапе не было вычислено не одной связи, то в зависимости от того, остались ли в формате невычисленные связи или нет, алгоритм завершится с ошибкой или успешно.

Такой подход позволяет разбирать форматы с циклическими зависимостями связей, которые семантически корректны. Пример приведен на рис. 10, где изображен формат с двумя связями A и B, которые меняются положениями в зависимости от внешнего параметра P.

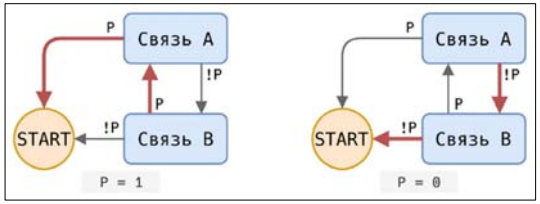

Рис. 10. Пример циклических зависимостей связей
Fig. 10. Relations cyclic dependencies example

Здесь их положения имеют циклическую зависимость на этапе компиляции, однако разрешаются корректно на этапе разбора. Реальным примером такого поведения является структура заголовка программы в формате исполняемого файла ELF (поля p_offset и p_flags).

Таким образом, порядок вычисления связей определяется динамически. Чтение самих данных из адресного пространства производится только в том случае, когда необходимо использовать это значение. В противном случае данные не читаются.

Алгоритм разбора обходит связи в порядке их указания в спецификации, поэтому в простейшем случае весь формат будет разобран за один проход. В худшем случае сложность работы алгоритма составляет $O(n^2)$, где n – количество связей формата. Оптимизация алгоритма, заключающаяся в построении графа зависимостей связей на этапе компиляции, позволяет достичь линейного времени работы алгоритма и запланирована в дальнейшей работе.

4.3 Язык Ribbit

Для спецификации форматов были реализованы необходимые синтаксические конструкции в языке *Ribbit*. Получившийся декларативный язык описания формата позволяет использовать всю выразительную мощность модели и одновременно является достаточно компактным за счет использования собственного синтаксиса.

В *Ribbit* в качестве численных переменных используются битовые вектора фиксированного размера:

```
let x: '16 = 0;
```

Битовые операции используют префиксную запись с апострофом в начале названия операции:

```
'add(1'32, 2'32).
```

```
data MyFormat(S: '64, E: endianness, F: data) {
    // Внутренняя связь
    magic: bytes(5, little),

    // Явное задание положения связи
    header: at(
        +(end(magic), 4),
        bytes(1, little)
    ) as '8,

    // Привязанное значение (битовый флаг)
    val flag: '1 = header[0],

    // Опциональное поле
    if flag {
        body: F,
    },

    // Связь с переменным форматом
    crc: if flag { CRC32() } else { CRC16() },

    // Внешняя связь
    extern tail: MyAnotherFormat(S, E),
}
```

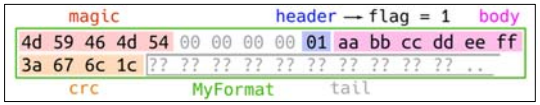

Рис. 11. Наверху: пример описания формата на языке Ribbit, внизу: пример разобранного буфера
Fig. 11. Top: example of format specification in Ribbit, bottom: example of parsed buffer

На рис. 11 приводится пример описания искусственного формата MyFormat на языке *Ribbit*, демонстрирующий возможности языка. Для тех полей, для которых не указано явно положение, компилятор автоматически выведет его. Поле tail соответствует внешней связи, поэтому его данные в буфере и размер обозначены как неизвестные.

В языке нет встроенной поддержки массивов, вместо этого используется вспомогательный формат list, представляющий собой структуру, схожую с односвязным списком (рис. 12). Возможность описывать битовые поля реализована через *Value*-связи типа битового вектора (см. поле val на рис. 11).

Разработанный алгоритм разбора имеет ряд ограничений, необходимых для завершимости. Например, конструкцию полей «тело + длина» (именно в указанной последовательности) невозможно разобрать, поскольку в ней имеются явные кольцевые зависимости: размер поля «тело» зависит от поля «длина», а положение поля «длина» зависит от положения поля «тело». В некоторых случаях (например, в описанном выше) компилятор способен выявлять такие конструкции и сразу выдавать сообщение об ошибке в спецификации формата. При этом описанный формальный язык сам по себе существует и данное сообщение об ошибке следует интерпретировать как то, что алгоритм не способен разобрать данный формат и гарантированно войдет в вечный цикл.

```
data list(len: '64, T: data) {
    if 'ne(len, 0) {
        head: T,
        tail: list('dec(len), T),
    }
}
```

Рис. 12. Описание массива на языке Ribbit
Fig. 12. Array specification in Ribbit

5. Примеры использования

На разработанном языке были описаны форматы:

- сообщения сетевого протокола DNS;
- исполняемого файла ELF;
- файла с растровой графикой PNG.

Хотя задача разработки оптимального по скорости разборщика не ставилась, были проведены сравнительные тесты времени разбора пакетов DNS трафика. Результаты представлены в табл. 2. Разборщик *Glassfrog* проигрывает по скорости другим решениям, однако в абсолютном значении показывает удовлетворительный результат. Стоит учесть, что для полностью корректного сравнения скорости работы следует уточнить требуемое выходное представление данных и детализацию спецификации протокола (необходимые поля). При этом в разборщике имеется широкий диапазон возможных оптимизаций, которые могут существенно увеличить его производительность. Например, с учетом вида представления форматов и результатов разбора, некоторые структуры результатов могут быть построены уже на этапе компиляции, поскольку имеют фиксированные размеры и адреса.

Табл. 2. Сравнительные тесты производительности разборщиков
Table 2. Parser efficiency comparison

Glassfrog (C API)	Kaitai Struct (C++)	Spicy (C++)
1,586,230 packets	1,891,769 packets	1,585,275 packets
7,412 packets/s	196,224 packets/s	52,156 packets/s
0,7 Kbytes/s	18 Mbytes/s	3,94 Mbytes/s

Разработанный язык отличается в том числе своей компактностью. Результат сравнения размера описаний форматов на различных языках[2] приведен в табл. 3. Для сравнения взяты инструменты Kaitai Struct и Spicy, поскольку они имеют в открытом доступе обширные базы описанных форматов данных.

Табл. 3. Сравнение количества строк в описаниях форматов
Table 3. Line number comparison in data formats specifications

	Glassfrog	Spicy	Kaitai Struct
DNS	57	89	214
ELF	66	91	331
PNG	39	35	94

6. Направления дальнейшей работы

На данном этапе работы реализация содержит ряд ограничений, обход которых является первостепенной задачей в дальнейшей работе.

6.1 Ограничения предлагаемого решения

Одной из главных трудностей при описании форматов данных с использованием *Ribbit* является описание полей, не выровненных по границам байтов. В случае использования форматом алгоритма сжатия или иного битового кодирования возникают сложности при спецификации. Например, при описании формата PNG, некоторые поля которого кодируются с помощью алгоритма сжатия gzip, возникает проблема с тем, что не удается вычислить общий размер поля. При этом использование битовых полей в качестве флагов покрывается возможностями *Value*-связей и взятием части от разобранного поля. Возможным решением этой проблемы является расширение множества встроенных форматов форматом битов.

Другой проблемой, возникающей при описании формата, является необходимость валидировать значения полей переменного размера. Частным примером является использование контрольных сумм, которые вычисляются от всего массива данных, который не имеет заранее известного размера.

Язык *Ribbit* имеет возможность описания функций, оперирующих с битовыми векторами фиксированной длины. Требование фиксированности длины здесь критично, поскольку компилятору необходимо проверять корректность используемых типов. Для описания функций с различными типами используются шаблоны, которые явно раскрываются в конкретные реализации функций.

В нашем случае длина битового вектора, для которого необходимо вычислить контрольную сумму, определяется динамически. Таким образом, компилятор не может сгенерировать определенную функцию даже с использованием шаблонов. Потенциальным решением может быть описание функции, принимающей указатель на данные (адресные пространства и указатели также поддерживаются в *Ribbit*).

При описании формата с полем в виде списка сущностей способом, описанным выше, с использованием формата list, разобранные данные представляют собой многократно вложенную структуру (рис. 13).

Такое представление не слишком удобно для пользователя, желающего оперировать с полем, как с массивом. Возможным улучшением будет использование inline-директив, которые

[2] Поскольку в различных языках существуют различные примитивы, брались частичные описания формата, сходные по структуре.

позволят явно подставлять поля одного формата в другие. В таком случае возможно реализовать обращение к элементам списка по индексам.

```
Struct(T)
  _: Struct(list)
      head [0..2] ------------------------------------------- 0x457f
      tail: Struct(list)
          head [2..4] ------------------------------------- 0x464c
          tail: Struct(list)
              head [4..6] ----------------------------------- 0x0102
              tail: Struct(list)
                  head [6..8] ------------------------------- 0x0001
                  tail: Struct(list)
                      head [8..10] ---------------------------- 0x0000
tail: Struct(list)
```

Рис. 13. Результат разбора формата списка list
Fig. 13. Parse result of list format

С целью упрощения спецификации форматов пользователем планируется разработка базы стандартных форматов, включающей в себя такие форматы как числа фиксированных размеров, списки, терминированные строки и т.п. Планируется также разработка инструмента для трансляции спецификаций распространенных форматов из других языков, обладающих обширными базами описанных форматов (например, из Kaitai Struct). С целью повышения производительности разборщика данных, планируется провести ряд оптимизаций реализации алгоритма: предварительное построение графа зависимостей связей, кэширование результатов вычислений выражений.

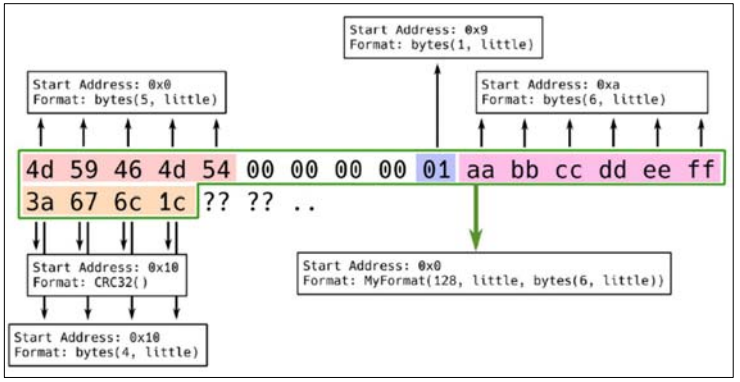

Рис. 14. Разметка буфера абстрактного состояния с использованием модели форматов
Fig. 14. Abstract buffer state markup using proposed data format model

6.2 Применение к задачам анализа бинарного кода

Одной из задач анализа кода, связанных с форматированными данными, является задача вывода типов данных. В процессе анализа бинарного кода частично восстанавливаются буферы памяти – определяются значения для некоторых адресов памяти. При дальнейшем изучении иногда удаётся определить, что конкретный буфер представляет собой не просто набор байтов, а вполне известную структуру, иными словами, имеет определенный формат. В таком случае информация об этом формате может быть полезна при дальнейшем анализе: анализируя последующие инструкции есть возможность отследить перемещение байтов

этого буфера, но теперь эти отслеживаемые байты будут представлять собой не просто помеченные данные, а данные определенного формата.

В простейшем случае, подобно анализу помеченных данных, можно отслеживать копируемые отрезки байтов и снабжать их разметкой формата. На рис. 14 приводится пример размеченного буфера с использованием модели форматов *Glassfrog*. Каждому биту сопоставляется экземпляр разобранных данных и смещение внутри него, соответствующее адресу бита. Разбор полей формата добавляет на те же самые биты новую разметку, использующую форматы полей. Эта разметка может использоваться как абстрактное состояние программы и с использованием абстрактной интерпретации [26] может быть решена задача вывода типов.

6.3 Применение к задачам генерации данных

Разработанная модель имеет потенциал для применения к задаче генерации данных. При генерации данных в соответствии со спецификацией может потребоваться генерировать как корректные данные, так и данные с ошибками (не соответствующие формату). Например, одним из видов ошибки может быть нарушение предиката корректности значения поля.

Для генерации корректных данных может быть модифицирован разработанный алгоритм разбора данных. В алгоритме чтение из пространства данных происходит только тогда, когда значение поля необходимо для вычисления некоторого выражения. В алгоритме генерации в этом случае будет необходимо сначала сгенерировать случайное значение, поместить его в пространство данных по требуемому адресу, а затем, также как в алгоритме разбора, прочитать его. Это позволит добиться корректности таких конструкций как «длина+тело». Поля, не использованные в выражениях, следует в конце заполнить произвольными данными. Такой подход не нарушит корректности структуры, однако может вызвать несоответствие предикатам корректности значений полей.

Для удовлетворения данным предикатам предполагается использование возможностей инфраструктуры Glassfrog, а именно символьного исполнения. Символьное исполнение позволит найти предикат на входные данные, удовлетворение которому будет обозначать истинность предиката корректности. Для него будет необходимо решить задачу выполнимости формулы (найти удовлетворяющие входные данные).

Символьное исполнение может также позволить изучать условные ветвления формата на предмет покрытия всех ветвей при генерации. Практические эксперименты с данными подходами запланированы в дальнейшей работе.

7. Заключение

В данной работе проводится анализ существующих подходов к декларативному описанию форматов данных и предлагается собственное решение, включающее модель формата и инфраструктуру для его спецификации. В ходе разработки были учтены выделенные ключевые особенности встречающихся форматов данных. Реализованный компилятор позволяет транслировать декларативное описание формата в представление, которое может быть использовано для задач разбора данных и совместного анализа кода и данных. Отличительными особенностями предлагаемого решения являются гибкость при задании положений полей формата, возможность конструирования и модификации представления с помощью API и, как следствие, возможность использовать модель для программного анализа форматированных данных. Разработанная инфраструктура была применена к спецификации распространенных форматов сообщений сетевых протоколов и исполняемых файлов.

Главными направлениями дальнейшей работы являются расширение модели для обработки битовых потоков и данных, не выровненных по границам байтов, и разработка механизмов вычисления функций от значений полей произвольного размера, в частности, для вычисления и валидации контрольных сумм.

Возможными прикладными приложениями модели к задачам анализа бинарного кода является использование представления при восстановлении форматов по трассе выполнения или образу программы, а также разметка данных памяти в задаче вывода типов.

Список литературы / References

[1] G. Back. DataScript - A specification and scripting language for binary data. Lecture Notes in Computer Science, vol. 2487, 2002, pp. 66-77.

[2] Хмельнов А.Е., Бычков И.В., Михайлов А.А. Декларативный язык FlexT - инструмент анализа и документирования бинарных форматов данных. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 239-268 / Hmelnov A.Y., Bychkov I.V., Mikhailov A.A. A declarative language FlexT for analysis and documenting of binary data formats. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 239-268 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-15.

[3] Kaitai Struct: declarative binary format parsing language. URL: https://kaitai.io/.

[4] McCann P.J., Chandra S. Packet Types: abstract specification of network protocol messages. ACM SIGCOMM Computer Communication Review, vol. 30, issue 4, 2000, pp. 321-333.

[5] Pang R., Paxson V. et al. binpac: a yacc for writing application protocol parsers. In Proc. of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC '06), 2006, pp. 289-300.

[6] Borisov N., Brumley D. et al. Generic Application-Level Protocol Analyzer and its Language. In Proc. of the Network and Distributed System Security Symposium, 2007, 16 p.

[7] Hopcroft J.E., Motwani R., Ullman J.D. Introduction to Automata Theory, Languages, and Computation. 3-е изд. Pearson, 2006, 560 p.

[8] Knuth D.E. Semantics of context-free languages. Mathematical systems theory, vol. 2, issue 2, 1968, pp. 127-145.

[9] Ford B. Parsing expression grammars: a recognition-based syntactic foundation. ACM SIGPLAN Notices, vol. 39, issue 1, 2004, pp. 111-122.

[10] Jim T., Mandelbaum Y., Walker D. Semantics and Algorithms for Data-Dependent Grammars. In Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2010, pp. 417–430.

[11] Afroozeh A., Izmaylova A. Iguana: A Practical Data-Dependent Parsing Framework. In Proc. of the 25th International Conference on Compiler Construction, 2016, pp. 267-268.

[12] Earley J. An Efficient Context-Free Parsing Algorithm. Communications of the ACM, vol. 13, issue 2, 1970, pp. 94-102.

[13] Jim T., Mandelbaum Y. A New Method for Dependent Parsing. In Proc. of the 20th European Conference on Programming Languages and Systems, 2011, pp. 378-397.

[14] Ganty P., Köpf B., Valero P. A Language-Theoretic View on Network Protocols. Lecture Notes in Computer Science, vol. 10482, 2017, pp. 363-379.

[15] Peach: a fuzzing framework which uses a DSL for building fuzzers and an observer based architecture to execute and monitor them. URL: https://github.com/MozillaSecurity/peach.

[16] Netzob: Protocol Reverse Engineering, Modeling and Fuzzing. URL: https://github.com/netzob/netzob.

[17] Sommer R., Amann J., Hall S. Spicy: a unified deep packet inspection framework for safely dissecting all your data. In Proc. of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 558-569.

[18] Fisher K., Mandelbaum Y., Walker D. The next 700 data description languages. ACM SIGPLAN Notices, vol. 4, issue 1, 2006, pp 2-15.

[19] Fisher K., Gruber R. PADS: a domain-specific language for processing ad hoc data. ACM SIGPLAN Notices,vol. 40, issue 6, 2005, pp. 295-304.

[20] boofuzz: Network Protocol Fuzzing for Humans. URL: https://github.com/jtpereyda/boofuzz/.

[21] GitLab Protocol Fuzzer Community Edition. URL: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[22] 010 Editor - Pro Text/Hex Editor. URL: https://www.sweetscape.com/010editor/.

[23] GNU poke, an extensible editor for structured binary data. URL: 10.5446/46118.

[24] Соловьев М.А., Бакулин М.Г. и др. Практическая абстрактная интерпретация бинарного кода. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 101-110 / Solovev M.A., Bakulin M.G. et al. Practical abstract interpretation of binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 6, 2020, pp. 101-110 (in Russian). DOI: 10.15514/ISPRAS-2020-32(6)-8.

[25] Соловьев М.А., Бакулин М.Г. и др. О новом поколении промежуточных представлений, применяемых для анализа бинарного кода. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 39-68 / Solovev M.A., Bakulin M.G. et al. Next generation intermediate representations for binary code analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 39-68 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-3.

[26] Cousot P., Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1977, pp. 238-252.

Приложение 1. Алгоритм разбора данных в соответствии с предложенной моделью (псевдокод)

```
    ВХОД: указатель на данные (экземпляр формата и адрес начала)
    ВЫХОД: структура результата Data

1. Сформировать множество неразобранных связей
2. Взять новую связь из множества неразобранных
3. Тип связи:
 - внутренняя или внешняя:
     3.1. Определить вычислимость положения
      - Не вычислимо: перейти к п.2
      - Вычислимо: вычислить -> POS
     3.2. Если POS = None, пометить связь как разобранную и
          перейти к п.2
     3.3. Определить вычислимость экземпляра формата:
      - Не вычислим: перейти к п.2
      - Вычислим: вычислить -> FORMAT
     3.4. Тип связи:
      - внутренняя:
         3.4.1. Для (POS, FORMAT) вызвать Алгоритм
         3.4.2. Полученный результат разбора добавить в структуру Data
         3.4.3. Пометить связь как разобранную
      - внешняя:
         3.4.4. Добавить (POS, FORMAT) в структуру Data
         3.4.5. Пометить связь как разобранную
      - связь-значение:
     3.5. Определить вычислимость значения:
      - Не вычислимо: перейти к п.2
      - Вычислимо:
         3.5.1. Вычислить -> VALUE
         3.5.2. Добавить VALUE в структуру Data
         3.5.3. Пометить связь как разобранную
4. Если еще есть связи в множестве неразобранных, перейти к п.2
5. Если ни одна связь не была разобрана, вернуть ОШИБКУ
6. Перейти к п.1
```

Информация об авторах / Information about authors

Александр Александрович ЕВГИН – аспирант ИСП РАН. Его научные интересы включают теорию формальных языков, форматы данных, сетевые протоколы, анализ бинарного кода.

Alexander Aleksandrovich EVGIN is a postgraduate at the system programming department. His research interests include formal language theory, data formats, network protocols, binary code analysis.

Михаил Александрович СОЛОВЬЕВ – кандидат физико-математических наук, старший научный сотрудник отдела компиляторных технологий ИСП РАН; старший преподаватель

кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают анализ бинарного и исходного кода, обратную инженерию ПО, операционные системы.

Mikhail Aleksandrovich SOLOVEV is a candidate of physical and mathematical sciences, senior researcher at the compiler technologies department of ISP RAS; senior lecturer at the system programming department of the faculty of Computational Mathematics and Cybernetics of MSU. His research interests include binary and source code analysis, software reverse engineering, and operating systems.

Вартан Андроникович ПАДАРЯН – кандидат физико-математических наук, ведущий научный сотрудник отдела компиляторных технологий ИСП РАН; доцент кафедры системного программирования факультета ВМК МГУ. Его научные интересы включают компиляторные технологии, безопасность ПО, анализ бинарного кода, параллельное программирование, эмуляцию и виртуализацию.

Vartan Andronikovich PADARYAN is a candidate of physical and mathematical sciences, leading researcher at the compiler technologies department of ISP RAS; associate professor of the system programming department of the faculty of Computational Mathematics and Cybernetics of MSU. His research interests include compiler technologies, software security, binary code analysis, parallel programming, emulation, and virtualization.