Автоматизация разработки моделей устройств и вычислительных машин для QEMU*

[1]*В.Ю. Ефимов <real@ispras.ru>*
[1]*А.А. Беззубиков <abezzubikov@ispras.ru>*
[1]*Д.А. Богомолов <bda@ispras.ru>*
[1]*О.В. Горемыкин <goremykin@ispras.ru>*
[1,2] *В.А. Падарян <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 им. М.В. Ломоносова, 119991, Россия, г. Москва, Ленинские горы, д. 1*

Аннотация. Разработка виртуальных устройств и машин для QEMU — трудоёмкий процесс. С целью поддержки разработчика, в данной работе был проведён анализ архитектуры QEMU и процесса разработки моделей отдельных устройств и виртуальных машин для QEMU. Предлагается подход к разработке, в рамках которого начальный этап ощутимо автоматизируется, благодаря применению декларативного описания устройств и машин, а также средств графического представления разрабатываемых устройств и машин. Подход реализован в интегрированном инструменте, позволяющем разработчику QEMU получить компилируемый набор файлов с исходным Си-кодом. Разработчик задаёт параметры генерации устройств и описывает состав машины на языке Python или в графическом редакторе, обеспечивающем визуализацию текстового описания. Результатом применения инструмента при построении машины становится фактически готовый Си-код, требующий только уточнить конфигурацию процессора и обработать параметры командной строки. В случае периферийного устройства от разработчика потребуется реализовать поведенческий аспект. Проведённые эксперименты с платформами Q35 и Cisco 2621XM показали, что количество строк в описании устройства в 11-26 раз меньше числа строк получаемой заготовки на языке Си. Такая разница в объёме достигнута за счёт генерации формального кода, реализующего служебные интерфейсы QEMU. Такой код составляет ощутимую долю кода устройства, в то время как может быть сгенерирован по сравнительно небольшому описанию. Суммарный объём сгенерированного кода заготовок составил от ¼ до ¾. Исходный код разработанного инструмента доступен по адресу https://github.com/ispras/qdt.

* Работа поддержана грантом РФФИ № 16-29-09632

Ключевые слова: программный эмулятор; бинарный код; разработка виртуальных машин.

DOI: 10.15514/ISPRAS-2017-29(6)-4

Для цитирования: Ефимов В.Ю., Беззубиков А.А., Богомолов Д.А., Горемыкин О.В., Падарян В.А. Автоматизация разработки моделей устройств и вычислительных машин для QEMU. *Труды ИСП РАН*, том 29, вып. 6, 2017 г., стр. 77-104. DOI: 10.15514/ISPRAS-2017-29(6)-4

1. Введение

Виртуальные вычислительные машины (VM) применяются для решения разнообразных задач: включая исследования в рамках информационной безопасности. Одной из задач является организация контролируемого окружения для исследуемого машинного кода во время динамического анализа. Устоявшийся подход совмещает дизассемблер IDA Pro и интерактивную отладку, когда сервером удалённой отладки выступает эмулятор. Эмулятор даёт дополнительный «рубеж» изоляции между исследуемым кодом и инструментами анализа [1]. Поэтому он хорошо подходит для исследования компьютерных вирусов и другого вредоносного ПО. Одним из программных средств для организации VM является эмулятор QEMU [2].

Эмулятор QEMU наиболее подходит для этой цели, поскольку обладает рядом поолезных свойств: полностью открытый исходный код (лицензия GPL), поддержка разнообразных гостевых архитектур (Intel x86, AMD 64, ARM, MIPS, PowerPC, SPARC и др.), реализация важных, с точки зрения динамического анализа, технологий и возможностей.

Если возникает необходимость в динамическом анализе машинного кода для узкоспециализированных процессорных архитектур или малораспространённых машин, то, скорее всего, готовой VM не существует. Разработка такой VM становится сама по себе существенной проблемой, поскольку в отсутствие полностью готовой VM анализируемый код до конца не работоспособен, а работоспособность кода необходима для итеративной отладки VM. В случае QEMU, чтобы приступить к итеративной работе над VM, требуется предварительно написать значительный объем служебного кода.

Даже самые простые машины состоят из десятков устройств. Ввиду огромного разнообразия физических устройств даже в обширной библиотеке виртуальных устройств QEMU редко удаётся обнаружить требуемое или совместимое устройство, особенно, когда речь идёт об узкоспециализированных машинах. В этом случае разработчик вынужден реализовывать большое количество виртуальных устройств. Несмотря на то, что QEMU архитектурно приспособлен к добавлению новых моделей, процесс их разработки — трудоёмкая задача.

Поскольку развитие эмулятора ведётся распределенным сообществом, вопрос создания инструментальной поддержки для разработчика VM не получает должного приоритета. Нередко новые машины на основе QEMU разрабатываются закрыто, для внутренних нужд компании. Применительно к «разовой» разработке VM создание инструментов поддержки не актуально и, более того, не целесообразно в силу отвлечения ограниченных ресурсов. Тем не менее, появление автоматизированных методов разработки VM и соответствующих инструментов было бы полезно для всего сообщества разработчиков QEMU.

Для создания такого инструмента был исследован процесс разработки VM и отдельных виртуальных устройств. Используемый в настоящее время подход заключается в поиске похожей функциональности в существующих моделях и реализации требуемой по образу и подобию. При этом применяется непосредственное копирование кода, с последующими его правками и дополнениями. Функциональное наполнение вносится согласно документации, а служебный код обновляется, чтобы соответствовать требованиям актуальной версии эмулятора. В данной работе предложен метод ускорения, основывающийся на выявлении и автоматизации рутинных этапов данного процесса. Метод был реализован в программном инструменте на языке Python [4].

Дальнейший текст организован следующим образом:

- рассматриваются работы, решающие схожие проблемы;
- описывается подход, используемый в QEMU для эмуляции отдельных устройств и целых машин;
- с учётом подхода к эмуляции формулируется предлагаемый подход к автоматизации разработки моделей;
- описывается разработанный программный инструмент, реализующий подход к автоматизации;
- предлагается процесс разработки устройств и VM с использованием разработанного инструмента;
- приводятся экспериментальные данные об использовании инструмента при разработке виртуальных машин Q35 и C2621XM.

2. Обзор похожих работ

Необходимость поддержать разработку новых VM актуальна для любого развивающегося эмулятора. Даже если рассматривается эмулятор с единственной гостевой процессорной архитектурой, для работы системного кода может потребоваться определённый комплект устройств, с определённой конфигурацией регистров ввода/вывода, прерываниями, взаимодействием с внешней средой и др. Среди множества известных эмуляторов стоит выделить такие проекты, как SimNow [5], Simics [6], gem5 [7] и OVPsim [8]. Первые два эмулятора — коммерческое ПО, gem5 — ПО с открытым исходным кодом,

распространяемым по лицензии BSD. OVPsim состоит из коммерческого ядра эмуляции и открытой библиотеки VM. Все перечисленные эмуляторы используются на практике и поддерживают быстрое добавление новых VM.

2.1 AMD SimNow

Эмулятор AMD SimNow [5] предназначен для упреждающей разработки низкоуровневого системного ПО для выходящих на рынок x86 процессоров AMD, в силу чего библиотека компонент VM содержит только процессоры этой фирмы. Создание новой VM происходит в графическом редакторе, где задаются связи между компонентами машины. Связь между устройствами задаётся пользователем, причём требуется указать пару имен интерфейсов у связываемых устройств. При задании связи сразу происходит проверка её корректности. У эмулятора SimNow имеется комплект разработчика (SDK), позволяющий создавать на языке Си++ как инструменты анализа (трассировщики и т.п.), так и новые устройства. Модели устройств в SimNow реализуются на базе иерархии классов Си++, которая в основном используется для наследования библиотечных методов.

Типизация устройств крайне проста. Как правило, базовый тип устройства отсутствует, наибольшая часть методов класса реализует служебную логику, необходимую для работы эмулятора. Пример исключения из такой практики — класс *CUsbMouse*, который наследуется от классов *CUSBDevice* и *CAutomationLib*. Первый базовый класс представляет абстрактное USB-устройство, второй — реализует возможность получать управляющие команды из консоли эмулятора для конфигурации устройства или от сценария инициализации. В SDK включены исходные коды типовых моделей устройств (ПЗУ BIOS, аудио- и видео- адаптеры, мосты различных шин, контроллеры прерываний и др.), на основе которых предлагается разрабатывать собственные модели.

2.2 gem5

В эмуляторе gem5 [7] предложена более сложная иерархия типов устройств. Еще одним отличием от SimNow, стала возможность быстрого прототипирования VM на языке Python, для которого была реализована привязка Си++ API. Реализация моделей устройств, требующих большого количества вычислений, ведётся на Си++, а их интеграция и задание конфигурации VM выносится в сценарий. Поскольку gem5 разрабатывается для детального моделирования современных процессоров, особенности работы которых влияют на производительность системного и прикладного ПО, большая часть библиотечных компонентов эмулирует многоуровневую память и топологию связей между вычислительными ядрами. «Медленные» периферийные устройства в gem5 фактически не представлены.

2.3 Simics

По сравнению с предыдущими двумя эмуляторами, Simics обладает наибольшими возможностями по созданию новых моделей машин и отдельных устройств. Как и gem5 объектная модель и API Си++ имеют привязку к языку Python. Помимо того, доступен специализированный язык DML (Device Modeling Language), предназначенный для моделирования устройств в Simics. Описание устройства на DML транслируется компилятором dmlc в Си++, из которого собирается разделяемая библиотека.

В DML программа описывает *класс-устройство*, возможно наследованный от некоторого базового класса. Полями класса-устройства могут быть объекты, которые должны представлять один из встроенных классов.

Расширять перечень встроенных классов (в документации для них используют термин *тип объекта*) пользователь не может. Встроенные классы описывают как базовые примитивы виртуальной аппаратуры, так и служебные данные, используемые эмулятором в работе. Базовые примитивы сводятся к двум понятиям: регистр и соединение. Имеется пять вспомогательных классов, используемых для различных способов группировки регистров и соединений. Класс (тип), называемый attribute, описывает произвольное свойство объекта, которое необходимо сохранять при создании снимка состояния. При добавлении к устройству полей регистров и соединений среда разработки автоматически создает для их описания поля типа attribute.

```
device excalibur;

connect bus {
    interface pci;
}
...
bank databank {
    parameter function = 1;
    register r1 size 4 @ 0x0000 {
        field f1 {
            method read { ... }
            method write { ... }
        }
    }
    register r2 size 4 @ 0x0004 {
        field f2 {
            method read { ... }
            method write { ... }
        }
    }
}
```

Рис. 1. Пример DML-описания модели устройства `excalibur`
Fig. 1. `excalibur` *device model DML description example*

На рис. 1 представлено сокращённое описание учебного устройства excalibur, подключаемого к шине pci. У устройства имеется *банк регистров*, в которых размещаются некие данные. Размер регистров — 4 байта, регистр r1 размещён в банке по нулевому смещению, регистр r2 — по смещению 4. Содержимое каждого регистра полностью покрывается единственным *полем*, для которого определены методы *чтения* и *записи*. Для описания семантики действий, происходящих в устройстве, тела DML-методов выражаются на языке, расширяющем подмножество ISO Си. Добавлены некоторые конструкции, характерные для Си++, такие как new/delete, try, throw.

2.4 OVPSim

Поскольку эмулятор OVPSim изначально рассчитан на привлечение сторонних разработчиков для создания новых VM, он предлагает открытый и обширный API на языке Си, который распадается на три части:

- VMI — разработка новых процессоров,
- PPM/BHM — разработка новых периферийных устройств,
- OP — интеграция компонент VM и контролирование её работы.

```
ihwnew -name simpleCpuMemoryUart
ihwaddbus -instancename mainBus -addresswidth 32
ihwaddnet -instancename directWrite
ihwaddnet -instancename directRead
ihwaddprocessor -instancename cpu1 \
    -vendor ovpworld.org -library processor -type or1k -version 1.0 \
    -semihostname or1kNewlib \
    -variant generic
ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA
ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 \
    -busslaveport sp1 -loaddress 0x0 -hiaddress 0x0fffffff
ihwaddmemory -instancename ram2 -type ram
ihwconnect -bus mainBus -instancename ram2 \
    -busslaveport sp1 -loaddress 0x20000000 -hiaddress 0xffffffff
ihwaddperipheral -instancename periph0 \
    -vendor freescale.ovpworld.org -library peripheral \
    -version 1.0 -type KinetisUART
ihwsetparameter -handle periph0 -name outfile -value uartTTY0.log \
    –type string
ihwconnect -instancename periph0 \
    -busslaveport bport1 -bus mainBus \
    -loaddress 0x100003f8 -hiaddress 0x100013f7
ihwconnect -instancename periph0 -netport DirectWrite -net directWrite
ihwconnect -instancename periph0 -netport DirectRead -net directRead
```

Рис. 2. Пример TCL-сценария для создания заготовки VM
Fig. 2. VM draft creation TCL script example

Ускорение разработки на начальном этапе обеспечивает утилита iGen (входит в SDK), которая по декларативному описанию на языке TCL генерирует заготовки. Утилита поддерживает автоматическое создание набора Си-файлов для моделей процессоров и устройств, встраиваемых в эмулятор; способна интегрировать интерфейс разрабатываемого Си-модуля с SystemC TLM2. Результат работы утилиты — полный комплект файлов с исходным кодом, которые компилируются в динамически загружаемую библиотеку. Описание определяет перечень регистров устройства, их отображение на память, интерфейсы шин. Поведение устройства реализуется в OVPSim через функции обратного вызова, сгенерированные файлы содержат объявления функций, а их определения имеют пустые тела, которые разработчик реализует самостоятельно.

На рис. 2 приведён пример TCL-сценария, описывающего простую VM с одним процессором OpenRISC 1000, 32-х разрядным адресным пространством, на два диапазона которого отображено ОЗУ, и регистрами UART, которые, в свою очередь, отображены на диапазон адресов с 0x100003f8 по 0x100013f7.

3. Разработанный подход к автоматизации

Рассмотренные в предыдущем разделе подходы к ускорению разработки предполагают использование декларативного описания интерфейсов устройств и всей VM. По описанию строится компилируемый код, представляющий собой заготовку устройства без реализации поведенческих функций или готовую VM, требующую незначительной «ручной» доработки. Для этого инструменты сборки дополняются транслятором описаний в язык Си/Си++ или любой другой, поддерживаемый эмулятором.

Целесообразно применять графические средства разработки для улучшения наглядности компонуемых VM. К сожалению, базовая версия QEMU до сих пор не предлагает никаких средств ускорения разработки VM. Их создание потребует учёта особенностей внутренней архитектуры данного эмулятора и технологических особенностей его разработки, которую ведет многочисленное распределённое сообщество.

3.1 Объектная модель QEMU

Основой использованного в QEMU подхода к моделированию машин и их компонент является «объектная модель QEMU» (англ. *QEMU Object Model* или, далее по тексту, *QOM*). Данная модель применяется не только для моделирования машин и их компонент, но и для реализации вспомогательных возможностей. QOM формирует иерархию *типов* (type), являясь реализацией парадигмы ООП на языке Си. Каждый тип имеет

уникальное строковое *имя*. Тип описывает *класс* (class) и *экземпляр* (instance). Экземпляров может быть много, в то время как класс один.

Иерархия QOM получается путём *наследования* (по аналогии с ООП) — установления отношения между двумя типами, таким образом, что один тип называется *ребёнком*, а другой — *родителем*. Ребёнок копирует всю информацию из родителя — *наследует*. Множественное наследование не поддерживается. Тип может быть *абстрактным*: такой тип не может иметь экземпляров.

В отличие от распространённых объектно-ориентированных языков, в QOM тип object (объект) не является корнем всей иерархии типов, а только одним из них. Данный тип добавляет *свойства* к экземплярам и классам. Свойство описывается строковым именем (уникальным в пределах объекта или класса), функциями доступа (присваивания (set) и разыменования (get)) и типом этого свойства. Со свойством связана и другая информация, но её рассмотрение выходит за рамки данной работы. Тип свойства ограничивает область его допустимых значений.

Моделирование VM и её элементов основано на потомках типа object. К ним относятся:

- машина (machine),
- устройство (device),
- шина (bus),
- запрос прерывания (irq),
- участок памяти (qemu:memory-region).

Модели VM, шин и устройств должны быть, соответственно, потомками machine, bus и device. Типы irq и участка памяти являются инфраструктурными, они используются в общем API, и их уточнение обычно не требуется. Дальнейшее наследование шин и устройств происходит по принадлежности к стандарту шины. При этом вводятся промежуточные типы, реализующие общий функционал. Конкретная модель устройства наследуется от типа соответствующего стандарту её шины. При реализации модельного ряда устройств добавляется промежуточный тип с общими для всего ряда особенностями.

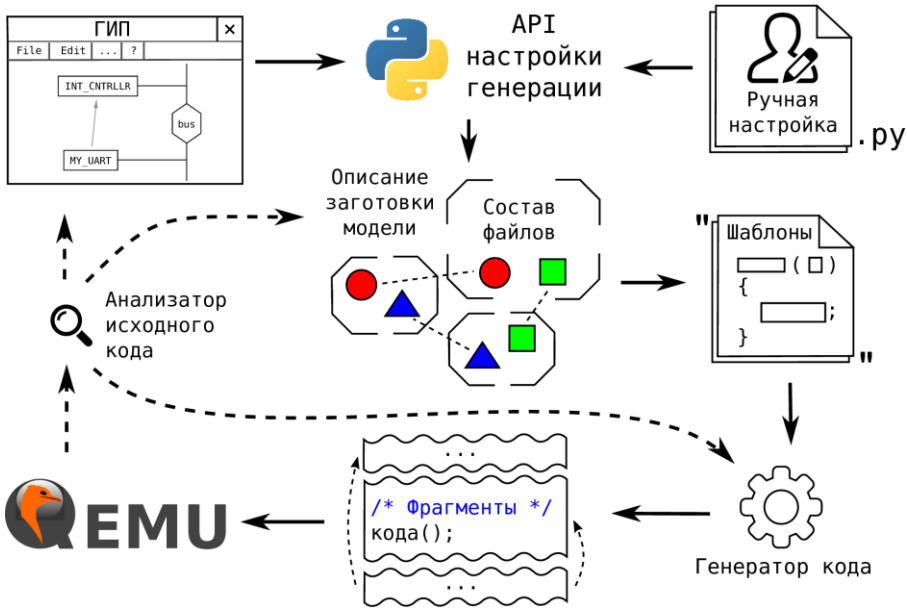

Рис. 3. Схема работы генератора заготовок моделей устройств и машин
Fig. 3. Machine & device draft generator system scheme

3.2 Метод автоматизации

На рис. 3 схематично изображён метод автоматизированной разработки эмулятора QEMU. Вручную или с помощью разработанного графического редактора разработчик задаёт основные параметры разрабатываемых моделей. На основе указанных параметров инструмент автоматически генерирует заготовки программного кода моделей. Эти заготовки не обладают законченной функциональностью, но готовы к компиляции.

Генерация основана на шаблонах — фрагментах программного кода, которые подстановками строковых параметров приводятся в синтаксически корректный код на языке Си. По исходному коду текущей версии QEMU генерируются необходимые включения заголовочных файлов и используемые макросы. Дальнейшая работа разработчика заключается во внесении в заготовку кода, соответствующего не формализованным особенностям работы устройства.

Метод шаблонов основан на следующей особенности кода QEMU. Любая модель, как устройство, так и машина, является частью эмулятора. Следовательно, её код, среди прочего, содержит фрагменты, обусловленные требованиями эмулятора (интеграция с QOM и т. д.), а не особенностями реального объекта, который эта модель описывает. Таким образом, условно

код модели можно разделить на две части: *индивидуальную* и *интерфейсную*.

В первую очередь рассмотрим это разделение для устройств. Индивидуальная часть задаёт поведение устройства в его программной реализации. Именно эта часть определяет то, как именно будет эмулироваться присутствие конкретно этого устройства в системе; делает его модель особенной, относительно других устройств, уникальной. При этом модель устройства является частью инфраструктуры QEMU, и индивидуальная часть должна эмулировать поведение устройства, используя возможности, предоставленные посредством API QEMU. В модели всегда можно выделить часть кода, взаимодействующую с этим API, она и называется *интерфейсной*. То есть, интерфейсная часть кода служит связкой между кодом индивидуальной части модели и остальным кодом эмулятора.

Индивидуальная часть обычно сформулирована на естественном языке в документации на устройство. При этом отсутствует единый формат формального описания, которого бы придерживались производители. Ввиду этого автоматизация разработки этой части весьма затруднительна и выходит за рамки этой работы.

С другой стороны, интерфейсная часть всех устройств очень похожа. Отличия заключаются, в основном, в перечне и количестве используемых моделью внешних интерфейсов, а также в именах собственных. Т. е. её параметры хорошо формализуемы.

Отдельно нужно сказать о применимости данного подхода к целой машине. В QEMU присутствует развитый API для интеграции устройств в единое целое, то есть в VM. Разработанный API во многом похож на API из QEMU. С помощью разработанного API можно формально описать полноценную VM. При этом имеются следующие ограничения.

1. Устройства, входящие в VM, должны иметь интерфейсную часть, реализованную в полном соответствии с принятым в QEMU подходом к написанию моделей устройств. Иначе несоответствие придётся компенсировать вручную.
2. Сгенерированная машина не поддаётся настройке, так как все её параметры зафиксированы на уровне исходного кода. Реализация возможности настраивать машины требует внесения кода вручную.

Первое ограничение не существенно, если преобладающая часть устройств VM реализуется вместе с ней по формальному описанию с помощью разработанного инструмента. Такие устройства будут иметь совместимую интерфейсную часть. Но в QEMU присутствует ряд устройств, которые были реализованы еще до того, как был выработан текущий подход к написанию моделей устройств. На данный момент не все из них были переписаны в соответствии с этим подходом. То есть они реализуют свою индивидуальную часть в обход новейших возможностей API для интерфейсной части. Если же

нужно использовать подобную модель устройства, то можно использовать разработанный инструмент, сгенерировав с его помощью новую интерфейсную часть, а затем скопировать индивидуальную часть из оригинала.

Второе ограничение заключается в следующем. Часть параметров машины обычно задается пользователем (а не разработчиком) только в момент запуска эмулятора через CLI (Command Line Interface):

- файл-образ ПЗУ (HDD, микросхема CMOS-памяти, CD, DVD, и т.п.),
- оконечную точку UART (виртуальный терминал, файл, и т.п.),
- способ подключения сетевого порта (TAP-адаптер, Ethernet по UDP и т. п.) и т. д.

Поддержка CLI требует внесения в код заготовки машины специального кода. Автоматизация этого процесса для наиболее часто используемых опций CLI является направлением дальнейших исследований.

3.3 API генератора заготовок

Состав интерфейсной части устройства определяется:

- обязательно используемыми служебными интерфейсами QEMU;
- потребностями индивидуальной части.

QEMU предоставляет ряд интерфейсов, из которых в интерфейсную часть выбираются только нужные. Код, соответствующий отдельно взятому интерфейсу, единообразен. Его можно получать из некоторого набора строковых заготовок, путём подстановки параметров. Часть параметров одних интерфейсов может быть связана с параметрами других интерфейсов. Часто для одного интерфейса необходимо сгенерировать несколько фрагментов кода. При этом фрагменты должны следовать в правильном порядке как относительно друг друга, так и относительно фрагментов других интерфейсов. Это требование следует из синтаксиса языка Си. Например, если один из аргументов функции является указателем на структуру, то сама структура должна быть объявлена выше. Кроме этого следует учитывать семантику фрагментов (близкие по смыслу фрагменты должны располагаться близко) и требования стиля программирования QEMU.

Генератор заготовок предоставляет API, упрощающий учёт этих и других особенностей. Состав API генератора условно можно разбить на две части:

- для *использования* шаблонов;
- для *добавления* шаблонов.

Каждый шаблон использует обе части. Через API использования шаблонов он добавляет себя в инфраструктуру инструмента, давая возможность применять себя для генерации. А с помощью API добавления шаблон реализует генерацию кода в соответствии с переданными ему параметрами.

3.1.1 Интерфейс использования шаблонов

Интерфейс использования шаблонов ориентирован на получение от пользователя перечня интерфейсов QEMU, требуемых разрабатываемой моделью, и их параметров. Для этого интерфейс использования предоставляет иерархию классов. Она построена по тому же принципу, что и иерархия классов QOM.

- *QOMDescription*
  - *SysBusDeviceDescription*
  - *PCIExpressDeviceDescription*
  - *MachineNode*

QOMDescription является базовым классом. Он не предназначен для использования. *SysBusDeviceDescription* описывает устройство на системной шине. *PCIExpressDeviceDescription* описывает PCI устройство QEMU.

Перечисленные классы выполняют роль контейнеров для параметров. Но, в то время как для описания устройств системной шины или PCI достаточно одного объекта перечисленных классов, для описания VM требуется ещё одна иерархия, рассмотренная ниже. Объект класса *MachineNode* служит контейнером для объектов той иерархии и прочих параметров генерации. Совместно объекты классов-потомков *QOMDescription* образуют *проект* (*QProject*), аккумулирующий данные для генерации кода. Классы, описывающие проект, устройства, VM и её состав, поддерживают сохранение этой информации в файл.

3.1.2 Модель вычислительной машины

Содержимое VM описывается с использованием следующей иерархии классов.

- *Node*
  - *BusNode*
    - *SystemBusNode*
    - *PCIExpressBusNode*
    - *ISABusNode*
    - *IDEBusNode*
    - *I2CBusNode*
  - *DeviceNode*
    - *SystemBusDeviceNode*
    - *PCIExpressDeviceNode*
  - *IRQLine*
  - *IRQHub*
  - *MemoryNode*
    - *MemoryLeafNode*
      - *MemoryAliasNode*
      - *MemoryRAMNode*

- *MemoryROMNode*

Node содержит уникальный идентификатор узла VM.

BusNode содержит все данные, описывающие шину любого типа. Все дочерние классы конкретизируют эти данные. Они были введены для сокращения объёма кода, необходимого для ручной работы с программным интерфейсом. При использовании графического редактора это не актуально.

DeviceNode описывает параметры самого устройства и настройки его интеграции. Классы, наследуемые от *DeviceNode,* расширяют этот список в соответствии с шаблонами устройств для конкретных стандартов шин.

IRQLine (линия) и *IRQHub* (концентратор) описывают распространение прерываний между устройствами (ту его часть, которая по какой-то причине не инкапсулирована в шину). Концентратор прерываний используется в случаях, когда одно прерывание должно быть доставлено в несколько устройств и/или может быть получено из нескольких устройств (так как линия прерывания соединяет строго два конца).

Следующие классы используются для явного создания участков памяти. Хотя большую часть адресного пространства машины определяют сами устройства, некоторые участки памяти, соответствующие ОЗУ, некоторым ПЗУ и т. п., должны быть добавлены явно. Для ОЗУ используется *MemoryRAMNode*, а для ПЗУ *MemoryROMNode*. *MemoryAliasNode* используется для ссылки на один участок адресного пространства из другого диапазона адресов. *MemoryLeafNode* (лист) является служебным промежуточным классом, запрещающим добавлять участки в участки, не являющиеся контейнерами.

3.1.3 Хранение настроек генерации

Объекты перечисленных выше классов объединены в проект (QProject). Формат файла, хранящего проект, основывается на возможности интерпретатора языка Python динамически добавлять код в программу. Сохранённый проект представляет собой код на языке Python. Поскольку инструмент сам написан на языке Python, данное решение существенно упростило разработку, поскольку не требует разработки специальных лексического, синтаксического и семантического анализаторов. Загрузка проекта представляет собой выполнение кода, в результате которого вырабатывается определение переменной, ссылающейся на объект класса *QProject*, который эквивалентен ранее сохранённому объекту. Генератор файлов разрабатывался таким образом, чтобы форматирование выводимого кода было удобно для человека.

Однако при сохранении *не* гарантируются:

- равенство *незначащих* пробельных символов;
- синтаксически незначащий порядок фрагментов кода (порядок определения аргументов конструкторов со значениями по умолчанию, порядок восстановления несвязанных объектов, и т.п.);

- сохранность имён переменных;
- использование тех же самых конструкций языка (например, программист может создавать несколько объектов в цикле, в то время как инструмент сгенерирует для этих объектов развёрнутый код);
- сохранность комментариев, если таковые были внесены вручную в файл проекта.

Эти особенности создают сложности при ручной работе с файлами проекта, и особенно при хранении файлов с использованием системы контроля версий. Решение этой проблемы является одним из направлений дальнейших исследований.

3.1.4 Способ генерации Си-кода и макрокоманд

Шаблоны реализованы с помощью форматных строк. Генерация кода заключается в подстановке параметров в форматные строки. Из полученных фрагментов кода затем составляются файлы. Форматные строки являются низкоуровневым способом определения шаблонов. Они ограничены в возможностях программной обработки. В связи с этим для разработки шаблонов был введён вспомогательный интерфейс. Будем называть его *интерфейсом добавления шаблонов*. Он основан на форматных строках, но предоставляемые им инструменты ориентированы на генерацию базовых конструкций языка Си. Поэтому интерфейс напоминает программную реализацию АСД, однако есть принципиальное отличие: он поддерживает и язык препроцессора.

Параллельно была предпринята попытка использовать для определения шаблонов АСД языка Си. Библиотека PyCParser [9], реализует двустороннее преобразование. Однако у применения АСД были обнаружены следующие недостатки:

- не поддерживается препроцессор, макросы которого активно используются в интерфейсной (и не только) части кода устройства в силу принятого в сообществе разработчиков QEMU стиля программирования;
- не учитываются незначащие символы (пробелы, переносы строк, комментарии), что требует доработки генератора PyCParser, чтобы он генерировал код, не противоречащий стилю программирования QEMU.

Основой интерфейса добавления шаблонов является модель *гибридного языка*, сочетающего подмножества языков Си и препроцессора. Выбор конструкций, которые попали в гибридный язык, обусловлен требованиями к оформлению кода QEMU, подробное рассмотрение этого вопроса выходит за рамки данной работы. Модель гибридного языка описывает содержимое заголовков и модулей языка Си. Разработчик описывает содержимое шаблона с точки зрения того, какие конструкции должны быть добавлены в файл в

соответствии с этим шаблоном. Причём один шаблон может касаться нескольких файлов, равно как и один файл может содержать конструкции из нескольких шаблонов. Некоторые конструкции из шаблонов связаны с уже существующими в QEMU конструкциями. Поэтому интерфейс позволяет определять содержимое существующих файлов. Описание существующих конструкций носит декларативный характер. Многие подробности могут быть пропущены, так как генерация не предполагается. Иными словами, используется принцип минимально достаточной информации.

При описании содержимого файла используются следующие классы.

- *Type*
  - *Structure*
  - *Function*
  - *Enumeration*
  - *Pointer*
  - *Macro*
  - *TypeReference*
- *Initializer*
- *Variable*
- *Usage*

Structure, Function, Enumeration и *Pointer* соответствуют структуре, функции, перечислению и указателю в языке Си. *Macro* используется для директив *#define* препроцессора.

Объект *TypeReference* (ссылка на тип) используется для ссылок на типы из других файлов. Любой тип может присутствовать непосредственно только в файле, где он объявлен. Но, когда один файл включается в другой с помощью директивы *#include,* второй косвенно содержит все типы первого. Чтобы отличить включённые типы от непосредственно определённых в данном файле применяется *TypeReference*. Ссылка создаётся для каждого включённого типа, включая те, которые уже были ссылками. Такой подход даёт возможность при генерации кода однозначно определить следует ли сгенерировать непосредственное определение типа по шаблону, или сгенерировать включение заголовочного файла, где он определён.

Одним из направлений дальнейших исследований является синтаксический анализ файлов QEMU с целью автоматического создания объектов, описывающих существующие типы. В настоящий момент такая функциональность реализована только для макросов препроцессора и опирается на функционал модифицированного препроцессора из библиотеки PyCParser.

Класс *Variable* (переменная) описывает пару: «тип, имя». Если к переменной нужно добавить начальное значение, то применяется класс *Initializer* (инициализатор).

Класс *Usage* (использование) применяется, когда требуется добавить инициализатор к *muny*, а не к переменной. Единственным примером в настоящее время является макрос. Инициализатор используется для расстановки значений при генерации вызова макроса.

Рассмотренная модель не является законченной. Но даже в таком варианте она позволяет с достаточной гибкостью описывать шаблоны устройств и машину, используемые при генерации интерфейсной части их кода. Развитие этой модели является направлением дальнейших исследований.

3.4 Генерация кода

Генерация кода заключается в получении из шаблонов фрагментов кода и объединении их в файлы. Причём полученные фрагменты должны быть упорядочены. Для этого была разработана вспомогательная модель файла с исходным кодом. Она оперирует неделимыми текстовыми фрагментами, из которых состоит файл, и порядковыми связями между ними. Элементы этой модели конструируются из элементов модели языка Си и препроцессора.

3.4.1 Модель файла с исходным кодом

Описанная выше модель языка Си и препроцессора не обеспечивает генерацию синтаксически корректного файла. Её задача: сгенерировать законченные *фрагменты* генерируемого файла. При этом остаётся решить следующие задачи:

- расположить фрагменты в синтаксически корректном порядке;
- обеспечить смысловую группировку фрагментов;
- соблюсти требования стиля программирования QEMU;
- минимизировать количество директив включения заголовков и др.

Далее эти задачи рассматриваются подробнее.

3.4.2 Сортировка фрагментов

Язык Си накладывает жёсткие ограничения на порядок определения различных символов. Например, тип должен быть объявлен до того, как будет создана переменная этого типа, или будет объявлена функция, принимающая аргумент такого типа. Подробное рассмотрение всех возможных примеров выходит за пределы данной статьи. Важно заметить, что почти все фрагменты связаны друг с другом, образуя ациклический граф, и для обеспечения синтаксически корректного порядка используется топологическая сортировка.

Помимо требований синтаксиса есть требования стиля программирования и здравого смысла, согласно которым, фрагменты должны следовать в следующем порядке:

1. включение заголовков;
2. объявление типов языка Си и макросов;
3. объявления функций;

4. определение функций, глобальных переменных и прочий код.

3.4.3 Учёт зависимостей и взаимосвязь с существующим кодом

Как уже отмечалось, для обеспечения видимости символов, объявленных в других файлах, генерируются директивы *include*. При этом имеется ряд тонкостей. Например, заголовочные файлы сами используют *include* для подключения других файлов, поэтому подключение одного файла может заменить подключение нескольких.

Эта особенность используется инструментом для сокращения количества подключаемых заголовков на основе анализа графа включения заголовков. Граф строится автоматически с использованием модифицированного препроцессора из библиотеки PyCParser.

Инструмент учитывает и другие особенности, рассмотрение которых выходит за рамки данной статьи.

3.4.4 Встраивание кода в QEMU

Для добавления заготовки устройства или платформы в QEMU кроме создания соответствующего исходного кода на Си, нужно зарегистрировать новые модули компиляции в системе сборки. Инструмент имеет соответствующую функциональность.

3.4.5 Адаптация к изменениям QEMU

QEMU является развивающимся проектом. Это приводит к тому, что в нём периодически происходят изменения, делающие шаблоны несовместимыми с новой версией.

Для решения этой проблемы используется эвристический подход. Все аспекты поведения инструмента, зависящие от версии QEMU, называются *эвристиками*. Так как один аспект работы может меняться многократно, то каждая эвристика представлена одной или несколькими записями в базе данных.

Код инструмента получает доступ к требуемой эвристике по *строковому ключу* — уникальному имени эвристики. Значением эвристики может быть любая сущность языка Python: от целочисленной константы до класса или модуля. Таким образом, при необходимости, можно *подменить* почти всю реализацию инструмента.

Каждая запись об эвристике имеет как минимум два значения: *новое* и *старое*. Запись привязывается к SHA1-идентификатору изменения в Git-графе [10] истории QEMU.

В инструменте реализован алгоритм, позволяющий для заданных SHA1, базы эвристик и Git-истории вычислить значения для всех имеющихся в базе ключей. Хранение обоих значений в каждой записи об эвристике избыточно. Но эта избыточность используется для проверки непротиворечивости записей.

3.5 Графический редактор

Все возможности инструмента доступны разработчику посредством интерфейса использования шаблонов. Для описания устройств и VM достаточно произвольного текстового редактора. Однако применение графического редактора, спроектированного специально для работы с этим интерфейсом, имеет следующие преимущества.

- Исключены лексические ошибки в именах переменных, названиях элементов интерфейса, а также синтаксические ошибки: разработчик вводит только значения параметров.
- Для многих значений параметров в QEMU определены макросы, использование которых предпочтительнее, согласно стилю программирования QEMU. Редактор, проанализировав код QEMU, может предоставить разработчику список доступных макросов, обычно применяемых с данным типом параметра. Например:
    - идентификатор PCI,
    - имя типа QOM,
    - список свойств выбранного устройства и т. п.
- Исключены некоторые семантические ошибки (например, в редакторе не предусмотрена возможность соединения линией прерывания двух шин, в то время как разработчик может написать подобное по ошибке). Имеется возможность дополнить редактор средствами поиска менее очевидных семантических ошибок.
- Все доступные параметры сосредоточены в *виджетах* и сопровождаются названиями на естественном языке. В большинстве случаев знания QEMU достаточно, чтобы понять суть параметра, не обращаясь к справочной информации.
- Интерпретация машины в виде схемы. Эта возможность особенно актуальна при разработке многоэлементных VM с большим количеством связей, так как на схеме легче ориентироваться, чем в тексте.

Для реализации графического редактора используется API Tkinter [11]. Данный интерфейс содержит всю необходимую функциональность и прост в развёртывании, так как включён в дистрибутивы Python.

3.6 Автоматизированный процесс разработки

С использованием разработанного инструмента процесс разработки как модели устройства, так и VM принимает следующую форму. Его можно разбить на 4 этапа: *ознакомление* с документацией, *подготовка* эмулятора, *генерация* заготовок интерфейсной части и *реализация* индивидуальной части.

При ознакомлении с документацией задача разработчика: оценить состав машины: какие устройства потребуется реализовать, как их связать в единую VM и т. д. Эта информация потребуется для генерации заготовок. Поскольку

инструмент генерирует заготовки с учётом текущей версии кода QEMU, может потребоваться внесение подготовительных изменений в код, чтобы задействовать максимум возможностей инструмента. Например, добавление новых идентификаторов PCI. Затем, разработчик задаёт параметры и получает заготовки — выполняет этап генерации. Наконец, он переходит к реализации индивидуальных частей кода устройств и уточнению заготовок VM. При этом происходит более детальное изучение документации, отладка и корректирование кода.

Проделанные в данной работе оценочные эксперименты проходили именно по такому сценарию.

4. Экспериментальные результаты

Для проверки состоятельности предложенного подхода с помощью разработанного инструмента были реализованы две VM:

- IBM PC совместимая машина Q35 на базе одноимённого набора микроконтроллеров фирмы Intel;
- маршрутизатор CISCO серии 2600 (C2621XM).

В качестве основной метрики эффективности автоматизации было выбрано количество строк. Подсчёты производились по истории Git и сгруппированы поэтапно. Замеры производились по разнице между начальной и конечной версией каждого этапа. Кроме этого для каждой VM приведена суммарная разница между базовой версией QEMU и полностью реализованной машиной. Важно иметь ввиду, что в приведённой ниже статистике изменение одной строки представлено как 1 удаление и 1 добавление в силу особенностей Git.

4.1 Q35

За основу Q35 была взята её реализация, уже имеющаяся в QEMU версии 2.9.5. Целью данного эксперимента была проверка возможностей разработанного инструмента. Выбор Q35 обусловлен тем, что эта машина является одной из самых сложных машин, реализованных в QEMU.

В ходе анализа исходного кода реализации Q35 был составлен перечень устройств и выявлена их взаимосвязь. Эти данные были формально описаны средствами разработанного инструмента. Полученная схема машины представлена на рис. 4.

Поскольку реализация Q35 не полностью соответствует всем требованиям QEMU, потребовался подготовительный этап. На этапе подготовки в QEMU вносились изменения, которые могут ограничить возможности инструмента:

- в заголовочный файл вынесена структура, описывающая устройство MC146818 RTC, и макрос динамического приведения к типу данного устройства;

- добавлены функции инициализации глобальных переменных *slave_pic* и *isa_pic*, в которые записываются ссылки на два устройства «i8259».

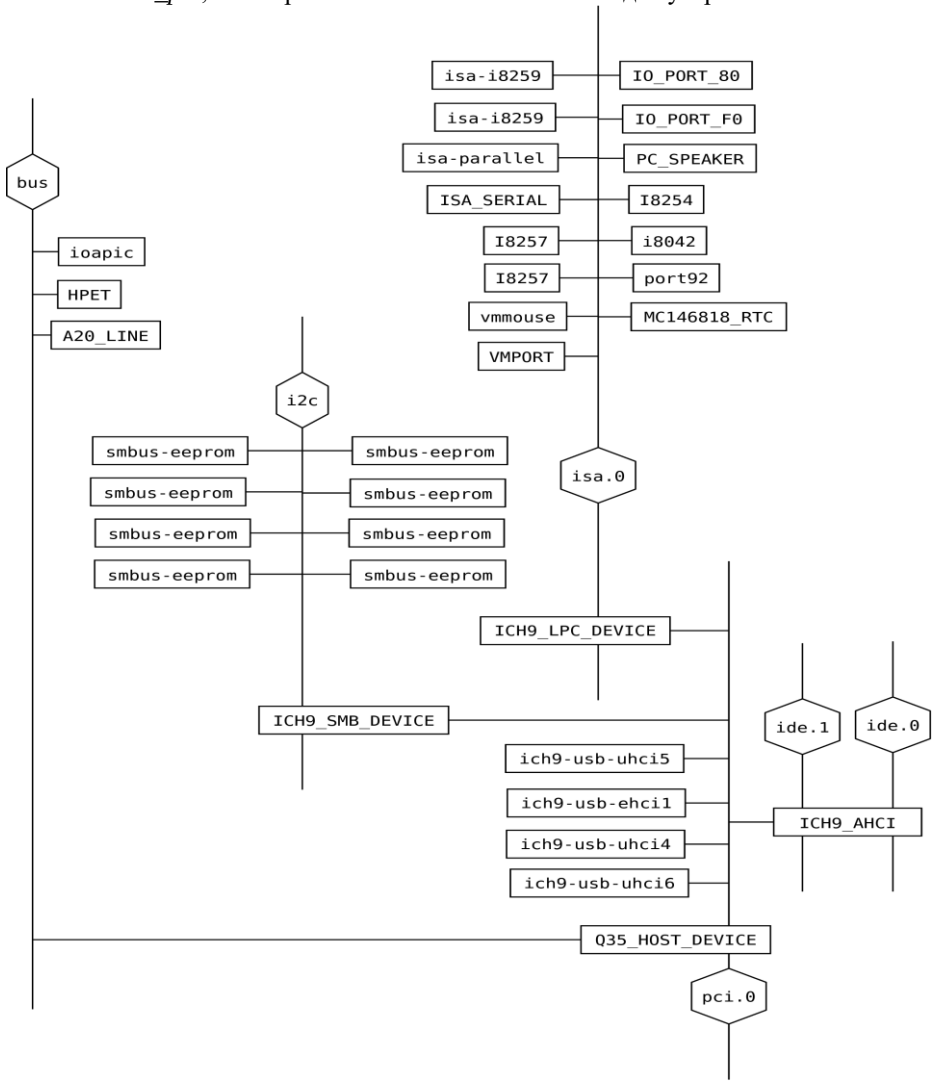

Рис. 4. Схема Q35
Fig. 4. Q35 scheme

В табл. 1 приведены оценки объёма работы по реализации Q35.

Табл. 1. Оценка объёма реализации Q35
Table 1. Q35 development steps evaluation

Этап	**Затронуто файлов**	**Вставлено строк**	**Удалено строк**
Подготовка	4	42	31
Генерация	8	599	0
Реализация	5	162	93
Суммарно	12	803	31

На этапе генерации с помощью инструмента были сгенерированы заготовки VM Q35 и трёх устройств, входящих в её состав: «A20 line», «port 80» и «port F0». В оригинальной VM Q35 эти устройства реализованы в коде самой машины.

На этапе реализации в заготовку VM были внесены следующие изменения:

- добавлен процессор;
- добавлена настройка BIOS;
- добавлен режим совместимости MS-DOS с исключениями FPU;
- выделена память под eeprom;
- добавлены к машине свойства, хранящие указатели на объекты устройств IOAPIC и PCI HOST;
- произведена инициализация ICH9 PM, IDE, PC CMOS, VGA, NIC и ACPI;
- скорректированы имена переменных.

Необходимость инициализации некоторых устройств на этапе реализации связана с тем, что эти устройства поддерживают настройку посредством CLI.

Стоит заметить, что в настоящее время инструмент не поддерживает группировку связанных по смыслу переменных в массивы и инициализацию их в цикле. Ввиду этого, инициализация прерываний была вынесена на этап реализации. Это позволило произвести регистрацию прерываний в цикле, тем самым обеспечив краткость кода VM. Инициализация прерываний заняла 20 строк кода, что составляет менее 3% от кода всей модели.

Таким образом, приблизительно 75% кода платформы было сгенерировано автоматически и лишь 11% пришлось модифицировать.

Итоговая версия платформы Q35 была успешно протестирована. Тестирование состояло из загрузки ОС Windows 7 и запуска Internet Explorer с последующим открытием страницы *google.com*.

4.2 C2621XM

За основу C2621XM взята его модель из эмулятора с открытым исходным кодом Dynamips [13]. Его разработка на данный момент заморожена, если не считать проект GNS3 [14], который использует Dynamips для эмуляции маршрутизаторов, коммутаторов и концентраторов, исправляя в нём ошибки.

На этапе подготовки в QEMU вносились следующие изменения:

- реализован MMU;
- реализована внутренняя коммутация прерываний процессора;
- исправлены некоторые регистры процессора специального назначения согласно с документацией [15];
- добавлены идентификаторы новых PCI устройств.

Первые два изменения позволили использовать процессор в режиме полносистемной эмуляции: до этого поддерживалась только эмуляция ABI ОС.

На этапе ознакомления, исходя из анализа исходного кода Dynamips, был составлен перечень устройств и выявлена их взаимосвязь. Схема VM представлена на рис. 5.

Начальной версией QEMU был выбран последний на тот момент выпуск 2.9.0. Все устройства, использованные в C2621XM, отсутствовали в QEMU и были перенесены из Dynamips. При этом все заготовки были сгенерированы с помощью инструмента. Стоит отметить, что устройства в Dynamips и, как следствие, их перенесённые в QEMU версии реализованы не полноценно, а лишь до той степени, чтобы удовлетворять потребностям некоторых версий системного ПО.

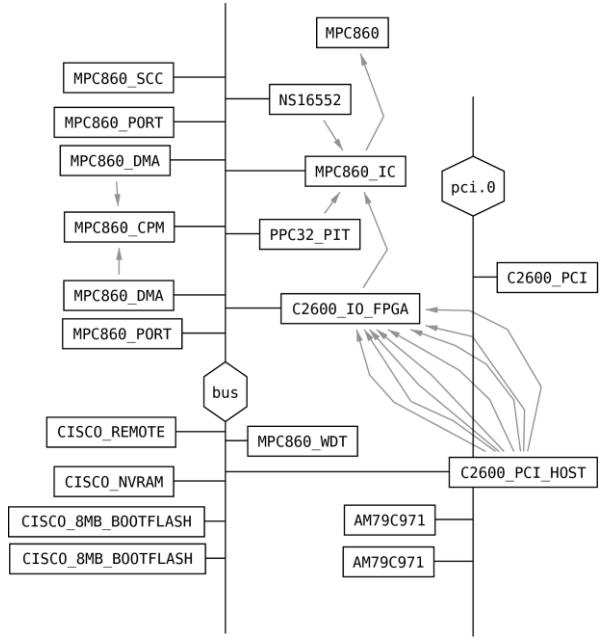

Рис. 5. Схема маршрутизатора C2621XM
Fig. 5. C2621XM router scheme

На этапе реализации была перенесена только индивидуальная часть устройств из Dynamips, а также скорректирована заготовка VM. Корректировка VM заключалась в следующем:

- связка параметров VM с CLI QEMU:
  - образы ПЗУ,
  - подключение сетевых интерфейсов,
  - подключение символьных устройств;
- реализация специального кода;
- корректировка имён переменных.

В табл. 2 приведена общая оценка объёма работы, а в табл. 3 дано сравнение количеств строк настроек генерации и полученных из них заготовок для некоторых устройств.

Табл. 2. Оценка объёма реализации C2621XM
Table 2. C2621XM development steps evaluation.

Этап	**Затронуто файлов**	**Вставлено строк**	**Удалено строк**
Подготовка	8	128	35
Генерация	37	2186	0
Реализация	31	4747	419
Суммарно	45	6642	35

Таким образом, приблизительно 1/3 кода всего маршрутизатора была сгенерирована автоматически. Причём только 1/5 часть сгенерированного кода потребовала модификации.

Табл. 3. Объёмы конфигурации и заготовок для устройств из C2621XM
Table. 3. Device generation configuration & resulting draft sizes for C2621XM

Устройство	**Конфигурация**	**Сгенерировано**	**Отношение**
MPC860_IC	6	125	20,8
C2600_PCI	7	82	11,7
AM79C971	12	175	14,6
NS16552	7	181	25,9

Итоговая версия маршрутизатора была успешно настроена для маршрутизации пакетов между двумя сетями (по одной на каждый интерфейс) и обеспечивала стабильное соединение в течении тестового времени (приблизительно 12 часов). В качестве генератора трафика использовалась утилита *ping*, настроенная на отправку ICMP запросов длиной 60кБ и 50кБ с машин из противоположных сетей. Сбоев замечено не было. Среднее время запроса составило 12мс.

5. Заключение

В ходе данной работы был исследован процесс разработки моделей устройств и машин для эмулятора QEMU. В изученном процессе был выделен рутинный начальный этап, хорошо поддающийся формализации. А именно, логику модели можно условно разделить на две части: индивидуальную и интерфейсную. Последняя служит прослойкой между индивидуальной частью и остальной VM, а также внешней средой. При этом интерфейсная часть содержит много формального кода, который может быть сгенерирован по сравнительно небольшому количеству параметров. Беглый анализ документации на устройство позволяет сформулировать эти параметры. Таким образом, в разработке устройства можно выделить промежуточный этап между изучением документации и реализацией. В течение этого этапа происходит генерация интерфейсного кода по заданному набору параметров.

Был разработан программный инструмент, автоматизирующий этап генерации. Инструмент поддерживает генерацию заготовок для устройств системной шины и шины PCI. Файлы настройки, используемые при генерации модели устройства, содержат в 11-26 раз меньше строк в сравнении с получаемой заготовкой. Этот подход может быть применён не только к устройству, но и ко всей VM в целом. Однако для VM подобной разницы в количестве строк достичь не удалось, ввиду того, что QEMU использует объектную модель для компоновки машин, что уже является шагом в сторону сокращения объёма кода.

По аналогии с этой объектной моделью в инструменте была разработана схожая модель, которая позволила представить VM в виде схемы в рамках графического редактора. Схема облегчает восприятие состава и связей VM для разработчика. Кроме того, имеется возможность применить малые автоматизации компоновки VM. Предложенный подход пригоден как для полной реализации VM вместе со всеми устройствами (при условии, что в QEMU уже есть поддержка соответствующей архитектуры процессора), так и для реализации VM с использованием уже имеющихся в QEMU устройств.

Инструмент был протестирован на двух VM: машина на базе Intel Q35 и машина C2600, являющаяся маршрутизатором фирмы CISCO серии 2600 (C2621XM). Они представляют две обозначенные крайности применимости данного инструмента, а именно, для Q35 в QEMU уже присутствовали все требуемые устройства, хотя некоторые из них пришлось привести в соответствие с требованиями объектной модели QEMU, используя данный инструмент. Для C2621XM, напротив, в QEMU отсутствовали все требуемые устройства. Доля сгенерированного кода составляет от ¼ до ¾ (в зависимости от количества уже реализованных устройств).

Перспективные направления исследований упоминались по тексту статьи. К числу наиболее важных из них относятся следующие направления.

- Автоматизация разработки поддержки процессорной архитектуры, что позволило бы полностью покрыть инструментом начальный этап разработки даже VM с новой для QEMU процессорной архитектурой.
- Поддержка добавления новых стандартов шин и генерации заготовок для устройств остальных, реже используемых шин, уже имеющихся в QEMU.
- Развитие графического редактора с целью внедрения в него малых автоматизаций, в совокупности облегчающих и ускоряющих процесс разработки.
- Реализация обратной связи о состоянии VM из запущенного QEMU с отображением информации времени выполнения на схеме машины, что позволит использовать инструмент ещё и для отладки в течение цикла итеративной разработки. Обратную связь предполагается организовать, запустив QEMU под отладчиком и контролируя состояние переменных времени выполнения и потока управления. Поскольку инструмент сам генерирует код VM, то информация, необходимая для сопоставления переменных времени выполнения и элементов схемы для него доступна.

Список литературы

[1]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9

[2]. F. Bellard QEMU, a Fast and Portable Dynamic Translator. USENIX Annual Technical Conference, FREENIX Track. USENIX, 2007. P. 41-465 p.

[3]. J. Bennett. Howto: GDB Remote Serial Protocol. Writing a RSP Server. Application Note 4. Issue 2, Embecosm, Доступно по ссылке: http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.pdf, ноябрь 2008.

[4]. Страница языка программирования Python. Доступна по ссылке: https://www.python.org/, дата обращения 06.07.2017.

[5]. Страница AMD SimNow Simulator. Доступна по ссылке: http://developer.amd.com/simnow-simulator/, дата обращения 12.10.2017.

[6]. D. Aarno, J. Engblom. Software and System Development using Virtual Platforms. Full-System Simulation with Wind River Simics. Elsevier Inc. 15.09.2014. 366p.

[7]. N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2, August 2011, 1-7pp.

[8]. Страница с документацией на Open Virtual Platforms. Доступна по ссылке: http://www.ovpworld.org/documentation/, дата обращения: 17.11.2017.

[9]. Страница библиотека PyCParser на GitHub. Доступна по ссылке: https://github.com/eliben/pycparser, дата обращения 20.03.2017.

[10]. Страница системы контроля версий Git. Доступна по ссылке: https://git-scm.com/about, дата обращения 09.03.2017.
[11]. Страница библиотеки для создания графического интерфейса «Tkinter». Доступна оп ссылке: https://wiki.python.org/moin/TkInter, дата обращения 2017.03.13.
[12]. В.Ю. Ефимов, К.А. Батузов, В.А. Падарян. Об особенностях детерминированного воспроизведения при минимальном наборе устройств. Труды ИСП РАН, том 27, вып. 2, 2015, стр. 65-92. DOI: 10.15514/ISPRAS-2015-27(2)-5
[13]. Руководство по работе с Dynamips. Доступно по ссылке: http://www.iteasypass.com/Dynamips.htm, дата обращения 30.06.2017.
[14]. Страница инструмента эмуляции сети GNS3. Доступна по ссылке https://gns3.com, дата обращения 30.06.2017.
[15]. Руководство пользователя для микроконтроллеров серии MPC860. Доступно по ссылке: http://www.nxp.com/docs/en/reference-manual/MPC860UM.pdf, дата обращения 30.06.2017.

Automation of device and machine development for QEMU

[1] *V.Yu. Efimov <real@ispras.ru>*
[1] *A.A. Bezzubikov <abezzubikov@ispras.ru>*
[1] *D.A. Bogomolov <bda@ispras.ru>*
[1] *O.V Goremykin <goremykin@ispras.ru>*
[1,2] *V.A. Padaryan <vartan@ispras.ru>*
[1]*Ivannikov Institute for System Programming of the RAS*
109004, Moscow, Alexander Solzhenitsyn st., 25
[2] *Lomonosov Moscow State University (MSU),*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Both virtual device and machine development for QEMU are difficult. To simplify the work of a developer we had analyzed both QEMU architecture and the development workflow. In this paper we suggest the new development approach which uses a declarative description for both machine and devices. The approach is implemented as an integrated software tool that returns a set of files containing a C code which could be compiled. Resulting code of machine is ready to use except for CPU configuration and CLI input. In case of a device, a developer has to implement the behavior of the device. Both device draft generation settings and machine content description are given to the tool in Python. A machine visual representation by a GUI is also implemented. A developer could use either GUI or a text editor (or both) to specify the settings. This way, the first stage of the development is automated. The tool was evaluated on Q35-based PC and Cisco 2621XM. The amount of device generation settings lines is 11-26 times smaller than the amount of the result code lines. This difference is achieved by generation of device model auxiliary code part which has a significant size because of QEMU API, while it could be generated using relatively small amount of settings. Generated code part is ¼ - ¾ of final machine code. The source code of the tool is available at https://github.com/ispras/qdt.

Keywords: software emulator; binary code; virtual machine development.

DOI: 10.15514/ISPRAS-2017-29(6)-4

For citation: Efimov V.Yu., Bezzubikov A.A., Bogomolov D.A., Goremykin O.V., Padaryan V.A. Automation of device and machine development for QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 77-104 (In Russian). DOI: 10.15514/ISPRAS-2017-29(6)-4

References

[1]. Dovgalyuk P.M., Makarov V.A., Padaryan V.A., Romaneev M.S., Fursova N.I. Application of software emulators for the binary code analysis. Trudy ISP RAN / Proc. ISP RAS, vol. 26, issue 1, pp. 277-296 (In Russian). DOI: 10.15514/ISPRAS-2014-26(1)-9

[2]. F. Bellard QEMU, a Fast and Portable Dynamic Translator. USENIX Annual Technical Conference, FREENIX Track. USENIX, 2007. P. 41-465 p.

[3]. J. Bennett. Howto: GDB Remote Serial Protocol. Writing a RSP Server. Application Note 4. Issue 2, Embecosm, Available at: http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.pdf, November 2008.

[4]. Python programing language site. Available at: https://www.python.org/, accessed: 06.07.2017.

[5]. AMD SimNow Simulator site. Available at: http://developer.amd.com/simnow-simulator/, accessed: 12.10.2017.

[6]. D. Aarno, J. Engblom. Software and System Development using Virtual Platforms. Full-System Simulation with Wind River Simics. Elsevier Inc. 15.09.2014. 366c.

[7]. N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2, August 2011, 1-7pp.

[8]. Open Virtual Platforms documentation web page. Available at: http://www.ovpworld.org/documentation/, accessed: 17.11.2017.

[9]. PyCParser library page at GitHub. Available at: https://github.com/eliben/pycparser, retrieved: 20.03.2017.

[10]. Git SCM. Available at: https://git-scm.com/about, retrieved: 09.03.2017.

[11]. Tkinter GUI library site. Available at: https://wiki.python.org/moin/TkInter, accessed: 2017.03.13.

[12]. V. Yu. Efimov, K. A. Batuzov, V. A. Padaryan, A. I. Avetisyan. Features of the deterministic replay in the case of a minimum device set. Programming and Computer Software, April 2016, Volume 42, Issue 3, pp. 174-186. DOI: 10.1134/S0361768816030038

[13]. Dynamips Manual. Available at: http://www.iteasypass.com/Dynamips.htm, accessed: 30.06.2017.

[14]. GNS3 network simulator site. Available at: https://gns3.com, 30.06.2017.

[15]. MPC860 chip series user manual. Available at: http://www.nxp.com/docs/en/reference-manual/MPC860UM.pdf, accessed: 30.06.2017.