1.4. ТЕХНОЛОГИЯ РАЗРАБОТКИ МУЛЬТИПЛАТФОРМЕННЫХ ПРОГРАММ НА ОСНОВЕ ЯВНЫХ СХЕМ ПРОГРАММ

Недоря А.Е., к.ф.-м.н.,
Zodiac Interactive

В статье обосновывается необходимость разработки мультиплатформенных систем – таких, части которых работают на разных платформах. Описана концептуальная схема таких систем.

Введение

В настоящее время все меньше смысла в разработке программ, работающих на одной платформе. Собственно, если задуматься, то практически любая современная программа взаимодействует с облачными серверами, сервисами обновлений, продаж и т.д., и является, таким образом, распределенной.

Это понимание не является общепринятым. И мы, как правило, разрабатываем программные системы по частям, используя разные среды разработки, языки и библиотеки на разных платформах, и не видя программную систему, как целое.

В лучшем случае, мы идейно остаемся на уровне кроссплатформенного программирования, то есть изготовления программ, которые могут быть запущены на нескольких платформах.

На мой взгляд, чтобы соответствовать современным требованиям, мы должны перейти на мультиплатформенное программирование, то есть на разработку распределенных программ, части которых работают на разных платформах. Естественно, при этом требовать, чтобы любая часть могла работать на любой платформе, обладающей достаточными ресурсами.

Упомяну несколько областей применения, которые трудно обеспечить программами без мультиплатформенной среды разработки:

- Разработка интернет-объектов для торгово-промышленно-финансового интернета, см. [1]
- Бытовое программирование [2], включая программирование IoT устройств
- И, может быть самое важное, обучение программированию, начиная с детского возраста – «программирование – вторая грамотность»

Постановка задачи

Около года назад мы поставили себе задачу создания экспериментальной технологии разработки мультиплатформенных программ, воплощенную в систему (среду) разработки «Вир-2», на основе ранее созданной среды разработки «Вир-1».

Говоря простыми словами, технология должна позволить разрабатывать мультиплатформенную программу, часть которой работает на сервере/в облаке, часть на мобильных устройствах, часть на специализированных устройствах (фитнес-браслет, смарт-часы), а часть на устройстве IoT. При этом разработка всех частей программы должна вестись в одной среде разработки, так, чтобы разработчик мог «почти» не обращать внимания на то, на каком устройстве части этой программы будут исполняться. Понятно, что нельзя не учитывать наличие специфичного оборудования и производительность, но все остальное должно быть несущественно.

Все части программы должны разрабатываться в рамках одного проекта и из одних и тех же компонент. При этом речь не идет о разработке нового языка программирования или о выборе одного из существующих языков.

Мы должны уметь использовать компоненты, написанные на любых языках, при условии, что они «совместимы» со средой разработки (или помещены в совместимую «обертку»). То есть речь идет не о переписывании всего, что уже сделано, а о некоторой объединяющей экосистеме, в которой могут использоваться сильные стороны и части уже существующего ПО.

Очевидно, что создание такой технологии и среды разработки является весьма сложным делом, поэтому надо использовать по максимуму все то, что уже есть.

В первую очередь, это наша среда разработки «Вир-1» [3] – среда сборочного программирования, которая показала свои достоинства в ходе 10-ти летней эксплуатации (подробнее ниже). «Вир-2» – это развитие среды «Вир» с добавлением нескольких новых свойств.

Во-вторых, не менее важным является использование наработок проекта LLVM [4], без которых разработка «Вир-2» в разумное время была бы невозможна.

В-третьих, мы учли опыт, полученный при изучении и использовании инструментов разработки кроссплатформенных программ (Xamarin, Marmelade, Lazarus) и кроссплатформенных библиотек (например, SDL2). Отдельно надо отметить экспериментальный проект Google NaCl (Native Client): технология запуска нативного кода в браузерах, основанная на LLVM и кроссплатформенном Pepper API.

Перед тем, как перейти непосредственно к описанию «Вир-2», опишем, на достаточном уровне, две основные составляющие части «Вир-2»: «Вир» и LLVM.

Среда разработки «Вир» – первая составляющая «Вир-2»

Около 10 лет назад мы начали делать «Вир», решая задачу ускорить и упростить разработку программ, а также повысить надежность за счет максимального использования сборочного программирования. Сборочное программирование позволяет собирать (большую часть) программы из стандартных компонент, дописывая недостающие и постоянно добавляя компоненты в репозиторий стандартных компонент.

Еще одно принципиально важное понимание, повлиявшее на «Вир» - это понимание того, что программа – это не одноразовое изделие, скорее это долгоживущий организм, который существенно модифицируется в ходе своего жизненного цикла. Соответственно, задача модификации программ, это более важная задача, чем начальная разработка. Отсюда несколько существенных требований к Виру с точки зрения модификации программ:

1. Иметь возможность максимально использовать то, что уже было сделано (нужна не только сборка, но и разборка программы на части в любой точке жизненного цикла),
2. Архитектурную целостность программы должна сохранятся в течении всего жизненного цикла,
3. Должно быть обеспечено упрощенное понимание программы за счет явной структуры. Уменьшая потребность в документации и избегая обычного расхождения между документацией и кодом программы и исходной архитектурой, и кодом.

Мы понимали экспериментальный характер разработки, так как идеи, которые проверялись, существенно выходили за рамки традиционных технологий программирования. И это, естественно, приводило к тому, что инструменты и подходы многократно менялись.

При этом основные идеи и подходы, заложенные в начале разработки среды «Вир», сохранялись неизменными, это:

1. Явная схема программы;
2. Сборка программ из бинарных компонент;
3. Репозиторий стандартных компонент;
4. Независимость программы от ОС.

Разберем подробнее эти идеи и их реализацию.

Явная схема программы

В большинстве случаев, современные программы в бинарной форме слабо структурированы. На стадии проектирования могут быть использованы различные структурные методы представления архитектуры. Далее, вручную или с помощью инструментов, строится исходный код, в котором архитектура размазана. И дальше, компилятор, особенно сильно оптимизирующий, удаляет оставшиеся следы архитектуры.

Как правило, из бинарной программы восстановить архитектуру программы невозможно. Более того, так как далее разработка продолжается, как правило, на уровне исходного кода, то соответствие между архитектурой программы и исходным кодом теряется полностью, в лучшем случае, частично.

Единственным средством явного структурирования программы являются DLL (SO), но использование DLL накладывает существенные ограничения на способы разработки и развития программ. Вплоть до того, что, например, в языке Go постулируется статическая сборка программ, чтобы не сталкиваться с проблемами динамической сборки.

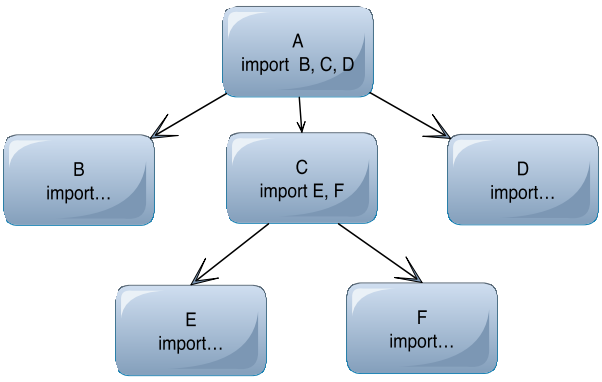

Рисунок 1. Пример модульной схемы, модуль A -– головной модуль программы

Альтернатива DLL существовала в реализациях модульных языков программирования, например, в системе Оберон [5] (язык реализации Оберон) или в OS Excelsior iV [6] (Модула-2). В обоих случаях, использовалась раздельная компиляция модулей и динамическая загрузка. При запуске программы, которая была представлена головным модулем программы, динамически подгружались используемые (импортируемые) модули, кроме тех, которые уже были загружены.

Схема работающей программы при этом сохранялась в виде таблиц импорта для каждого модуля. Схему можно было извлечь из бинарных образов модулей и симфайлов, в которых хранилась информация об экспорте/импорте каждого модуля.

К сожалению, вместо развития модульных языков, которое могло привести к устранению их недостатков, индустрия пошла по пути использования слабоструктурированных языков, ярким примеров которых является C++.

Вернемся к схеме программ, характерной для модульных языков. Для них схема – это дерево, корнем которого является головной модуль программы, а переходы к узлам – это использование модуля (импорт), рис. 1.

Перечислим недостатки модульной схемы:

1. Устройство схемы или направление роста дерева. Корень дерева в модульной схеме – это модуль верхнего уровня, то есть модуль, обращенный к пользователю (для нас не важно, пользователь – это человек или другая программная часть). А это приводит к тому, что при любом изменении функциональности, корень дерева меняется. И это еще полбеды, гораздо хуже, что головной модуль становится узким местом при изменениях программы. Гораздо естественнее для «живой» модифицируемой программы схема, в которой корень находится внизу, и из него разворачивается функциональность.

2. Статичность схемы. Во-первых, изменение схемы делается только добавлением импорта в модуль и компиляцией. Динамическое изменение/построение новой схемы невозможно. Во-вторых, для использования компоненты необходимо при разработке иметь доступ к описанию её интерфейса. А если интерфейс изменился, то необходима перекомпиляция. Казалось бы, что этот недостаток преодолевается использованием средств ООП – описанием базового класса и наследованием (и еще нужна фабрика объектов или подобные механизмы). Да, но только частично. Хотя бы базовый класс должен быть описан во время разработки использующей компоненты. А при использовании базового класса мы постоянно сталкиваемся с типичной проблемой CLOP (class-oriented programming) языков – изменение в базовом классе приводит к перекомпиляции всех наследников. Еще более важно, что как только добавляются классы, теряется простота и ясность модульной схемы.

3. Следующий недостаток модульной схемы, о котором подробнее будем говорить позже, это то, что все узлы в модульной схеме одного и того же уровня. Понятно, что модули могут быть разного размера, но это не создает вложенность. В схеме программы должна присутствовать вложенность иерархий (например, дерево деревьев), иначе применимость схемы будет существенно ограничена.

Реализуя явную схему программы в среде «Вир» нам удалось избежать перечисленных выше недостатков модульных схем. Рассмотрим подробнее.

Устройство схемы в среде «Вир»

Схема программы в «Вире» – это дерево деревьев, корень программы – это самая низкоуровневая компонента, содержащая инструменты программы, необходимые для запуска программы. Далее разворачивается дерево компонент высокого уровня. Компоненты высокого уровня мы называем «рабочими столами». При этом каждый рабочий стол – это дерево компонент. Дерево обращено к пользователю кроной. Ближе к корню располагаются более «сервисные» компоненты. Ближе к кроне компоненты, более ориентированные на пользователя, не важно, кто является пользователем – человек либо программа.

Принципиально важно то, что схема отделена от компонент. Схема первична, компоненты вторичны. Если у нас есть схема программы, то любую компоненту можно заменить на другую без изменения или компиляции. Программа продолжит работать, если эти компоненты, старая и новая, «совместимы» (о совместимости надо говорить отдельно).

Рассмотрим пример схемы программы, рис. 2. На верхнем уровне схема состоит из рабочих столов (РС). Каждый рабочий стол определяется поддеревом.

Схема в «Вире» позволяет добавлять функциональность не только статически (во время разработки), но и «на лету», например, за счет скачивания дополнительных компонент из облака. Так можно добавить как

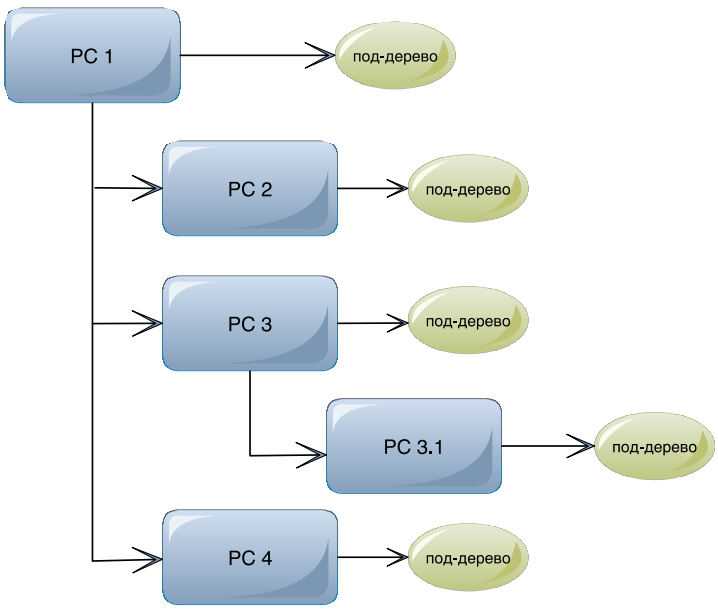

Рисунок 2. Пример схемы программы

PC, так и компоненту в под-дерево PC.

Динамический способ используется, например, для реализации механизма «дополнений» (add-ons), используемых в программах, сделанных в «Вире». Причем, этот механизм работает не для конкретной программы, а может быть применен в любой из создаваемых программ.

Взаимодействие компонент

Ранее упоминался один из недостатков модульной схемы: статичность схемы и статичность подключения компонент через импорт. Этого недостатка нет в «Вире».

Каким образом реализовано взаимодействие компонент в «Вире»? Способ связи компонент, то есть способ, каким компонента подключаются к тем компонентам, которые нужны для её работы, пожалуй, есть принципиальная ноу-хау часть нашей технологии.

Мы не используем статические механизмы подключения, типа import/include. Связь строится на взаимном расположении компонент в схеме (в иерархии).

Как правило, для того, чтобы компонента a могла использовать компоненту в, используется настройка компоненты a через относительный адрес. Настройка записывается в нотации привычной для адресов файлов, например, "./β", хотя семантика перехода несколько отличается. В некоторых случаях используется абсолютный адрес или поиск по тегам, но в этой статье мы не будем их рассматривать.

Как правило, если компонента a использует компоненты β и y, то, скорее всего, компоненты β и y будут находиться над ней в дереве – ближе к корню, т.к. дерево схемы мы рассматриваем растущим вниз.

Но если компонента β используется не только компонентой α, а другими компонентами, например, компонентой δ, то компонента β будет сдвинута вверх (к корню дерева), так чтобы быть выше компонент α и δ.

При запуске или при первой необходимости компонента обращается по адресу и получает доступ к используемой компоненте. Первое обращение является динамическим, далее компонента может запомнить адрес (аналогично получению адреса функции из DLL или SO) и далее обращаться напрямую или каждый раз выполнять динамическое обращение. Выбор между способами – это ответственность разработчика компоненты. Для увеличения гибкости, взаимодействие компонент может быть задано на простом скриптовом языке.

Очевидным преимуществом такого способа связи является возможность разработки независимых и заменяемых компонент. Возможный недостаток – это потеря производительности на критических задачах. Но наш подход дает возможности оптимизации, которые были ранее не применимы, которые, на наш взгляд, компенсируют потери. Впрочем, разговор об этом выходит за рамки статьи.

Сборка программ из бинарных компонент

Следующей особенностью «Вира», хотя и не такой необычной, как явная схема программы, является использование при сборке программы компонент в бинарной форме, а не в форме исходных текстов.

Использование бинарных компонент является принятым для component-oriented programming, например, BlackBox [7].

В случае «Вира», если все компоненты, необходимые для программы уже сделаны, программа может быть собрана без программирования. В реальной жизни, всегда есть функциональность, которую имеющиеся компоненты не обеспечивают, поэтому требуется разработка новой компоненты. После изготовления компоненты, она рассматривается с точки зрения полезности для других программ. И помещается в репозиторий стандартных компонент или в репозиторий компонент программы.

Замечу, что я намеренно использовал глагол «сделать» во фразе: «нужно сделать новые компоненты». Здесь скрывается еще одно существенное отличие от других систем разработки, использующих компоненты. В репозитории «Вира» хранятся компоненты разных уровней, которые существенно отличаются по устройству и способам изготовления.

Компоненты низкого (первого) уровня, по сути, являются заранее скомпилированными функциями. Очевидные примеры: поиск по строке или быстрая сортировка. Мы называем такие компоненты «словарными статья», очевидная аналогия – слово в языке Форт. Такие компоненты пишутся на языке программирования и компилируются в код, к которому прилагается описание интерфейса. Они используются при изготовлении компонент более высокого уровня и не могут быть напрямую использованы в схеме программы (в отличие от Форта, где нет ничего кроме «слов»).

Замечание: в традиционном программировании полезные «словарные статьи» принято собирать в библиотеки. В 50-х годах прошлого века понятия «библиотека» было открытием. В 1967 г. Морис Уилкс (Maurice V. Wilkes) получил премию Тьюринга за совокупность достижений, включающих в том числе концепцию программных библиотек.

Еще более любопытно, что мы используем слово библиотека (library) для очень разных сущностей, в том числе и для того, что вовсе не похоже на «библиотеку» в традиционном понимании.

Если использовать аналогию с книжной библиотекой, библиотека должна содержать только функции без состояния (stateless), что позволило бы использовать каждую (экспортированную) функцию (книгу) отдельно. Пример такой чистой библиотеки: библиотека математических функций.

С другой стороны, если нам нужна одна книга (например, синус), то зачем подключать библиотеку целиком, а потом, за счет использования статического линкера убирать неиспользуемые функции (если, конечно, библиотека не динамическая)?

В «Вире вместо библиотек используются словарные статьи, то есть если в программе нужен синус, то будет подключена словарная статья «синус», и все статьи, которая эта статья использует (если такие есть). И ничего больше. Для статей используется статическое связывание.

Компоненты второго уровня, мы называем «инструментами». Инструмент программируется и состоит из специфичного кода и обращений к словарным статьям. Об инструменте можно думать как об объекте или модуле, хотя эта аналогия не совсем точна. Инструмент – это минимальная единица, которую можно подключить к схеме программы. Особым видом инструментов являются «столы» или «верстаки», которые служат для объединения инструментов в компоненту более высокого уровня. Инструмент, как и словарная статья, компилируется один раз и затем используется в бинарной форме.

Компоненты третьего уровня – это «узлы», которые строятся из инструментов и столов. Пример: текстовый редактор.

Узел – это составная компонента, которая является атомарной (черным ящиком), с точки зрения сборщика программы. С другой стороны, разработчик узла видит её как белый ящик и может разобрать и собрать по-другому. При изготовлении узла уже не используется компилятор, разве что для изготовления недостающих инструментов. Устройство узла определяется схемой соединения его компонент.

На этом иерархия компонент не заканчивается, например, есть еще рабочие столы и дополнения, но рамки статьи не позволяют нам говорить о них подробнее.

Репозиторий стандартных компонент

Репозиторий позволяет накапливать компоненты, которые будут использоваться в следующих программах. Он состоит из несколько слоев:

1) Базовый слой – самые универсальные компоненты. Например, то, что относится к обязательным составляющим программ (логирование, обновление программы, защита, поддержка UI);
2) Тематический слой – раздел разбит на подразделы, например: Локальная сеть, Конструктор сайтов и т.д.;
3) Слой кандидатов на попадание в базовый или один из тематических слоев.

На каждом слое хранятся компоненты всех уровней.

Независимость программы от ОС

Несмотря на то, что «Вир» делался как среда разработки для одной платформы (Windows), мы, с самого начала, считали, что изготавливаемая программа должна быть изолирована от ОС.

И далеко не только потому, что кроссплатформенная разработка была в планах. Но, главное, потому что использование API любой конкретной ОС имеет свою специфику, которая, если не принять меры, просачивается в код программы, искажая архитектуру.

Поэтому, изоляция от ОС – это не только способ повышения переносимости, но и архитектурной чистоты.

Как сделана изоляция? В репозитории есть словарные статьи, которые позволяют обратиться к функционалу ОС. С точки зрения разработчика инструментов эти статьи ничем не отличаются от остальных, а при исполнении они обращаются к компоненте «Переходник», которая переводит эти обращения в вызовы одной или нескольких функций ОС. Реализация «Переходника» различная для разных платформ.

Словарные статьи переходника добавлялись по требованию, то есть мы не пытались полностью закрыть ОС (например, Win32), а исходили из потребностей разрабатываемых программ.

История разработки среды «Вир» и результаты

Приведем историческую справку развития проекта среды разработки «Вир»:

- начало разработки среды – 2006: конструктор программ и компилятор;
- первая программа, сделанная в «Вире» – «Картмейстер» с началом продаж в 2007 году;
- в «Вире» сделан конструктор сайтов «Сайткрафт» – начало продаж 2008 г., сейчас в продаже 11-я версия программы «Сайткрафт-Студия»;
- проверка подхода на различных прикладных задачах (несколько десятков программ) и доработка среды – до 2016;
- сейчас «Вир» (5-я версия) используется для разработки «Вир-2» и быстрого прототипирования в рамках проектов ООО «Синергетик Лаб»;
- размер репозитория компонент на текущий момент: 1200 компонент 1-го уровня, 4400 компонент 2-го уровня, 200 компонент 3-го уровня.

Статистика использование компонент в последних разработках:

- «DoReMind» – прототипирование перед разработкой мобильного приложения.

Из 194 компонент, используемых в программе, 184 было взято из репозитория, разработано 10 новых компонентов. Процент использования готовых компонент: 95%.

- «Анализатор текстов» для проекта «Лекса»

Из 176 компонент, используемых в программе, 172 было взято из репозитория, разработано 4 новых компоненты. Процент использования готовых компонент: 98%. Заметим, что программы существенно отличаются по назначению друг от друга и от тех программ, которые делались до этого в «Вире».

«DoReMind» – программа конвертирующие данные от нейро-интерфейса в мелодические ритмы.

«Анализатор текстов» используется для сбора статистики использования словоформ (слоги, слова) по заданному набору текстов (необходимо для выстраивания последовательности обучения чтению детей с дислексией).

Использование «Вира» позволило сделать программы быстро с небольшими затратами времени/ресурсов. Принципиально то, что работа шла в уютном режиме как для команды разработчиков (разработчик, дизайнер, музыканты для DoReMind, логопеды и педагоги для «Анализатора текстов»). Цикл: изготовление прототипа, тестирование, уточнение требований, изготовление нового прототипа занимал не недели, как в случае традиционных систем разработки, а дни или часы. Важно и то, что разработчик, в основном, думал не о том, как сделать, а о том, что сделать.

LLVM – вторая составляющая «Вир-2»

Так как, возможно, не все хорошо знакомы с проектом LLVM, приведу краткое описание с официального сайта (llvm.org): "The LLVM Project is a collection of modular and reusable compiler and toolchain technologies".

Любопытно, что изначально название проекта «LLVM» было аббревиатурой от "Low-Level Virtual Machine". Но проект ушел так далеко, что исходное название ему совсем не подходит. LLVM теперь – это не аббревиатура, а полное название проекта, как бы странно оно не звучало без гласных. Заметим, что подход авторов к названию хоть и, несомненно, оригинальный, но не дотягивает до оригинальности рекурсивных аббревиатур. Например: GNU, как известно, это рекурсивная аббревиатура GNU's Not Unix. Впрочем, llvm звучит вполне похоже на команды внутреннего представления (LLVM IR), которое является основой всего проекта, например: lshr или cmpxchng. Так что, на самом деле, чувство вкуса авторам LLVM не изменило.

Чтобы от шуток перейти к серьезному разговору, напомню, что в 2012 году основные разработчики LLVM (Chris Lattner, Evan Cheng, Vikram Adve) были удостоены премии ACM Software System Award, которая присуждается только одной программной системе в год.

Что же все-таки такое LLVM? Говоря простым языком, LLVM – это набор технологий, библиотек и утилит для построения компиляторов, оптимизаторов, верификаторов и других подобных программ. Основой LLVM является внутреннее представление – LLVM IR (Intermediate Representation) [8].

С использованием LLVM для реализации любого языка программирования на любую, поддерживаемую проектом платформу (а это все современные платформы), достаточно разработать только анализатор (front-end), который строит IR. Далее, достаточно подключить оптимизаторы и генераторы кода, входящие в LLVM Core Library.

Известным примером использования LLVM для построения компилятора является Clang (LLVM native C/C++/Objective-C compiler), который, собственно, и есть анализатор, строящий IR, и, далее, запускающий оптимизатор и генератор кода для целевой платформы.

Кроме C/C++/Objective-C LLVM используется в компиляторах языков Ruby, Python, Haskell, Java, D, PHP, Pure, Lua и других.

Как архитектор многоязыковой компилирующей системы XDS (eXtensible Development System) [9] я могу оценить высокое качество решений, заложенных в основу LLVM. В том числе, качество IR, уровень которого, на мой взгляд, удачно выбран (не слишком высокий и не слишком низкий). В XDS, в качестве внутреннего представления, использовалось абстрактное синтаксическое дерево, что приводило к сложностям из-за слишком высокого уровня промежуточного языка.

LLVM существенным образом упрощает разработку мультиплатформенных программ, так как избавляет от необходимости искать решения для генерации кода на разные CPU.

Вир-2

После того, как рассмотрели составляющие части, вернемся к разговору о технологии разработки мультиплатформенных программ.

В «Вир-2» мы используем отработанную технологию сборки программ, с очевидными изменениями, связанным с использованием LLVM. Перечислим их кратко:

- На этапе разработки словарные статьи переводятся в LLVM IR (а не в код CPU)
- Каждый инструмент переводится в IR для специфического кода со ссылками на IR для используемых словарных статей
- На этапе сборки готовой программы, для каждой кодовой части проверяется наличие и актуальность кода в кэше для конкретной платформы и при необходимости выполняется генерация кода по IR.

Соответственно, в репозитории хранится IR, а готовый код компонент кэшируется для ускорения сборки программы.

Работа, связанная с переходом на LLVM является достаточно простой с точки зрения понимания. Существенно более сложной является задача по созданию среды исполнения, которая позволит исполнять полученные программы (части программ) на разных платформах.

Очевидно, что:

- Среду исполнения **нельзя сделать, её можно только делать**, добавляя новое, и убирая совсем старое;
- Для разных предметных областей среда приложения будет частично пересекаться, а частично различаться;
- Среда исполнения должна легко расширяться;
- В ней должны быть разные варианты интерфейсов, не должно быть единственной альтернативы — только так и не иначе.

То есть, мы изначально должны говорить не о наборе неких функций или модулей (POSIX, WIN32, NaCl Pepper), а о том, **каким способом должен строиться «переходник» (abstraction layer)** между приложением и базовым слоем софта на устройстве.

Мы полагаем, что переходник не должен быть монолитом, а должен собираться (автоматически) под каждое приложение (часть приложения) с грануляцией на уровне функций. Некоторый функционал добавляется в переходник, если он нужен для работы хотя бы одной компоненты приложения.

На первом этапе мы предполагаем в «Вир-2» создание сред исполнения для трех платформ: Windows, Linux, Android. Для реализации используются насколько возможно кроссплатформенные библиотеки, покрывающие нужную функциональность. После проверки работы среды «Вир-2» на этих платформах, мы планируем увеличить количество платформ.

Обучение программированию

Одно из потенциальных использований «Вир-2», которое мы собираемся активно продвигать – это обучение программированию, начиная с детского возраста – «программирование вторая грамотность».

Принципиальная особенность Вира – это сборка, которой можно обучать раньше, чем традиционному программированию. Упомянем, в качестве примера, Scratch-2 – среду визуального сборочного программирования для детей.

В отличие от систем, подобных Scratch-2, мы собираемся сделать открытую среду разработки для детей, то есть среду, позволяющую использовать окружающие детей устройства. Частично об этом говорится в [2].

Любопытно, что использование Вир-2 позволит ребенку оставаться в одной среде разработки.

Если привести аналогию, то ребенок сначала делает детскую машинку из LEGO, а потом собирает настоящий автомобиль из настоящих деталей. В виртуальном мире это возможно, как и возможна программа, которая начата, как система управления детской машинкой, а потом развивается до управления автомобилем или космическим кораблем.

Переходя от аналогии к Вир-2: ребенок сначала собирает программу из готовых компонент в визуальном редакторе, а потом переходит к программированию компонент и сборке их любым удобным для себя образом.

Заключение

Разговор о технологии разработки мультиплатформенных программ должен быть практическим. В целом мы понимаем, что комбинация проверенных технологий, подходов и инструментов позволит сделать систему разработки мультиплатформенных программ. Но только на практике можно убедиться в качестве решения и в том, что технология даст существенный прирост скорости разработки и качества программ. Задачу проверки подхода мы решаем разработкой «Вир-2».

Разработка «Вир-2» носит экспериментальный характер, и мы намерены выкладывать результаты наших экспериментов в открытый доступ для того, чтобы желающие могли подключится к работе, как с критическими замечаниями, так и с практическим вкладом.

Для начала, мы выкладываем ознакомительную версию «Вир-1» [10]. Для получения версии, напишите заявку на сайте: http://vir.synergetic-lab.ru

Литература

1. Недоря А.Е., Буняк В.В. "Интернет — в поиске чистого воздуха" // http://digital-economy.ru/stati/internet-v-poiske-chistogo-vozdukha, 2017
2. Недоря А.Е. "Забытое 40 лет назад новое, и как оно может изменить нашу жизнь" // Сборник трудов SoRuCom-2017, М.:ФГБОУ ВО «РЭУ им. Г. В. Плеханова», 2017, стр. 243-250.
3. Недоря А.Е. «Вир» // заметки в блоге http://алексейнедоря.рф/?cat=13
4. The LLVM Compiler Infrastructure, http://llvm.org/, 2017
5. N. Wirth and J. Gutknecht. Project Oberon. Addison-Wesley, 1992. 488 p

6. Кузнецов Д.Н., Недоря А.Е., Тарасов Е.В., Филиппов В.Э. КРОНОС: семейство процессоров для языков высокого уровня. Микропроцессорные средства и системы, 1989.
7. Warford, Stanley. Computing Fundamentals. The Theory and Practice of Software Design with BlackBox Component Builder, 2002. 611 p
8. LLVM Language Reference Manual, http://llvm.org/docs/LangRef.html, 2017
9. Недоря, А. Е., Расширяемая переносимая система программирования, основанная на биязыковом подходе : диссертация кандидата физико-математических наук : 05.13.11. - Новосибирск, 1993. – 139 с.
10. Вир-1, сайт поддержки // http://vir.synergetic-lab.ru/

Недоря Алексей Евгеньевич (aleksei.nedoria@synergetic-lab.ru)

Ключевые слова

технология разработки программ; мультиплатформенные программы; сборочное программирование; схема программы; бинарные компоненты.

Nedorya A.E. A technique for the multiplatform software development based on the explicit software schema.

Keywords

Software development, multiplatform software, assembly programming, software schema, binary components.

Abstract

The article underpins the devepolment of multiplatform software. Parts of the software in context operate on different platforms. A conceptual schema of the software is discussed.

DOI: 10.34706/DE-2018-02-04