Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Создание библиотеки для функционального реактивного программирования роботов на базе платформы .NET

Курсовая работа студента 344 группы
Кирсанова Александра Юрьевича

Научный руководитель
к.ф.-м.н. Я. А. Кириленко

Санкт-Петербург
2014

Содержание

1 Введение

1.1 Расширение диапазона доступных программных средств

Современные смартфоны и специализированные платы отстают по скоростям и объёмам памяти от обычных десктопных компьютеров всего на пару лет. Это позволяет переходить от классического для микроконтроллеров “C” к языкам с более богатыми средствами выразительности и современным технологиям, применение которых накладывает на скорость определённые издержки, недопустимые в микроконтроллерах с маломощными процессорами. Кроме использования новых платформ и продуктов для встроенных систем, которые позволяют расширить круг решаемых задач и упростить разработку, становится возможным применять технологии, разработанные и для персональных компьютеров. Это помогает ускорить процесс разработки за счёт увеличения уровня абстракции, с которой приходится работать программисту.

Кроме того, повторное использование популярных технологий в сфере производства ПО имеет ещё одно очень важное преимущество – это отсутствие временных издержек и прочих затрат на её внедрение, создание или перенос необходимой инфраструктуры и сторонних библиотек. Существующие продукты не нуждаются в проверке на состоятельность и в обучении сотрудников новым техникам. Таким образом будущий успех использования существующей технологии зависит только от степени её применимости в новом контексте и от сложности адаптации технологии под новые платформы и задачи.

1.2 Роботы

Робототехника в наши дни является точкой соединения целой группы самостоятельных научных областей. Алгоритмы, которые робот использует во время своей работы неразделимо связаны с большим количеством вычислений. А сложность и объемы программ для современных роботов не уступают коммерческим разработкам в остальных сферах производства ПО и требуют правильной организации проекта для создания расширяемых, повторно используемых программ. Поэтому применение проверенных средств для создания промышленного ПО особенно актуально для робототехники.

1.3 Роботы проекта Трик

TRIK – Кибернетическая платформа, разрабатываемая компанией КиберТех на кафедре системного программирования СПбГУ. На контроллере установлен специальный дистрибутив Linux, учитывающий специфику всевозможных портов и работу с цифровым сопроцессором и процессором, отвечающим за взаимодействие с высоковольтными моторами. Образ

ОС содержит все необходимые драйверы для работы с устройствами, представленными в наборе TRIK. В контексте решаемой задачи, стоит отметить, что все новые прошивки кроме специфичного программного обеспечения содержат устрановленный Mono 3.4, что позволяет исполнять .NET assemblies. И вести разработку на всех .NET совместимых языках.

1.4 Задача

Исследование доступных средств и существующих концепций и их последующее применение для создания библиотеки функционального реактивного программирования роботов на базе фреймворка .NET.

2 Используемые технологии и актуальность разработки

2.1 Mono

Mono — проект с открытым исходным кодом, реализующий платформу .NET на системах отличных от Windows (начиная от Linux, MacOS, Android и заканчивая Sony Playstation и Wii). В настоящее время позволяет использовать достаточно полное подмножество функциональности всех версий целевого фреймворка для многих популярных архитектур (x86/64, PowerPC, SPARC, ARM). Кроме компилятора C# и среды исполнения, Mono предоставляет широкие средства для оптимизации и профилирования, что помогает более тонко следить за процессом загрузки и проводить анализ узких мест непосредственно на роботе.

Открытие исходных кодов многих продуктов Microsoft, активное использование Mono для разработки как десктопных, так и современных мобильных кросс-платформенных приложений, говорит о только большем внедрении .NET в современную разработку программного обеспечения. Поэтому использование .NET на встроенных системах осмысленно и обещает быть логичным шагом в развитии программных средств, используемых на микроконтроллерах.

2.2 Программирование на F#

F# - мультипарадигменный язык программирования, берущий свои корни в ML семействе. В 2010 году Microsoft открыла исходные коды всех связанных с F# проектов, в настоящее время поддерживается open-source сообществом. Существует компилятор и интерпретатор языка для разных ОС.

По своему устройству язык F# очень похож на OCaml и по сути является реализацией этого языка поверх инфраструктуры .NET, однако имеет и существенные отличия. В F# отсутствует такая мощная работа с модулями и полиморфные варианты. Преимуществами F# является полная совместимость с объектной инфраструктурой .NET, что позволяет активно использовать существующие для фреймворка библиотеки и создавать на F# новые.

Кроме этого в F# появились активные паттерны, единицы измерения, цитирование кода, но, пожалуй самым важным, преимуществом является появление computation expressions, эквивалента монад из Haskell. Через них на языковом уровне естественным образом выражаются такие мощные средства как асинхронные и ленивые вычисления. Асинхронное программирование активно использовалось при создании этой библиотеки.

3 Реактивное программирование

Существует большое количество программ, работа которых неразрывно связана с активным взаимодействием с внешним миром. Эти взаимодействия могут заключаться в ожидании получения ответа на запрос, который был отправлен в сеть, или обработкой движений мыши или нажатия клавиш. Чаще всего, например при программировании GUI, программа не должна переставать быть доступной при работе пользователя с диалоговым окном и параллельно выполняются сразу несколько задач.

Такие приложения можно условно назвать data-oriented. Вся логика работы в таких программах заключается в обработке данных, а удобство написания таких приложений очень сильно зависит от того какими идеями и средствами мы будем пользоваться при её создании. Парадигма программирования, которая хорошо подходит для написания такого типа приложений называется Реактивной.

Реактивное программирование оперирует с потоками данных, позволяя легко модифицировать эти потоки, создавать новые, которые являются сложной композицией других, при этом распространяя все последующие изменения в исходных потоках через преобразования в результирующий поток.

Принципы реактивного программирования, как нельзя кстати применимы и к роботам. Которые по своей сути являются очень реактивными системами. Роботы проводят большую часть своего времени, получая данные от все возможных датчиков, при этой обработке робот должен быстро реагировать на те изменения которые произошли. Например робот квадрокоптер выполняет задачу стабилизации. Предположим для простоты, что он получает данные только от гироскопа. На основании отклонения от горизонтального положения роботу необходимо поменять мощность, которую он выдает на моторы пропеллеров.

Существуют библиотеки для разных языков программирования, реализующие похожую функциональность. Поэтому при разработке библиотеки реактивного программирования роботов важно учитывать доступность таких средств как существующие библиотеки и то, насколько сам язык идейно близок к реактивному программированию.

События в F# привносят функционал, аналогичный предлагаемому сторонними библиотеками для языков без языковой поддержки событий и реактивной парадигмы. Кроме того для .NET реализован мощный аппарат для реактивного программирования в виде отдельной библиотеки Rx (Reactive Extensions), которая представляет потоки как Observable-последовательности и предлагает множество средств для работы с ними.

3.1 F# Events

События в F# – first class value, мы можем передавать события аргументами в функции, Существует модуль Events, в котором реализованы базовые методы для работы с событиями единообразно спискам и другим контейнерам. Тем не менее использование Событий и Observable последовательностей кардинально отличается моделью предоставления данных. Списки, массивы, всевозможные очереди и стеки реализуют интерфейс IEnumerable<T>, который использует модель вытягивания данных (Pulling data). Для обращения к определенному элементу необходимо явно указать, что мы хотим его получить.

Observable последовательности реализуют IObservable<T>. При использовании Observable, мы следуем модели выталкивания данных (Pushing Data). Вы один раз указываете, что вы хотите получать данные от этого источника, производя про этом некие формальные действия. Все дальнейшие изменения, происходящие с источником, будут автоматически поступать к вам и должным образом обрабатываться.

Использование Событий в F# имеет большой недостаток в виде проблем с отпиской после агрегаций. Обычные F# события не безопасны в плане освобождения памяти. Если есть последовательность событий, которая модифицируется Event.Map, Event.zip, Память занимаемая исходной последовательностью может быть не освобождена.

3.2 Observable-последовательности и Reactive Extensions

Rx[6] – библиотека, нацеленная на работу с асинхронно появляющимися данными в event-based приложениях. При этом заменяя абстракцию меняющего объекта на поток событий, каждый элемент которого является состоянием объекта во времени. При изменении объекта к потоку присоединяется его новое состояние. С потоками данными, которые могут приходить с задержками, но отсутствие которых не должно останавливать работы приложения, приходится сталкиваться в разработке веб-приложений, GUI и при долго выполняющихся вычислениях.

Основной принцип работы Rx может быть представлен через два интерфейса: IObservable<T> и IObserver<T>. По существу они являются продолжением идей известного паттерна Наблюдатель[2]. Observable последовательность – данные, которые приходят с некоторыми задержками, при этом при появлении нового элемента, сигнала о завершении или ошибки все объекты, которые “наблюдают” за нашей последовательностью будут “оповещены”. IObservable предоставляет ровно один метод Subcsribe, через который

будет осуществляться подписка. Для стандартных структур предусмотрено большое количество перегрузок этого метода, начиная от callback'ов, вызов которого будет осуществляться при появлении нового элемента, до консолидированного объекта IObserver, обладающего реализацией для возникновения ошибок и правильной обработки завершения потока событий. .NET обладает средствами, которые можно использовать для аналогичных задач. Прежде всего это Multicast Delegats, но делегаты и события (которые на самом деле тоже являются делегатами), в отличие от Observable, не обладают обработкой завершения и возникновения ошибок, но ещё более важными преимуществом Rx является активная работа с потоками. События не предоставляют средств для упрощения работы в многопоточных приложениях (в случае, когда события одного потока, обрабатываются на другом)

IObservable , IObserver включены в состав .NET начиная с версии 4.0 вместе с минимальным набором методов, позволяющих работать с ними как с другими контейнерами. Reactive Extensions не входит в .NET, но может быть легко установлен с помощью пакетного менеджера. В Rx реализовано огромное количество средств для агрегации Observable-последовательностей, работой со временем и управлением потоками. IObservable является дуалом к IEnumerable. Все методы, реализованные для работы с перечислимыми контейнерами, добавлены к IOservable и наоборот. Все эти возможности, вместе с гибким переходом к любым другим контейнерам (списки, массивы, ...), и использованием IScheduler превращает Rx в мощный аппарат для реактивного программирования.

4 Реализация библиотеки

4.1 Общие замечания об архитектуре

Библиотека создавалась с сильным прицелом на использование совместно с Rx в ключе реактивного программирования, кроме этого проект очень сильно опирается на внутреннюю архитектуру Контроллеров TRIK и используемой в них периферии. Однако открытый исходный код, библиотеки сможет облегчить создание/перенос аналогичного функционала на другие платформы с похожим внутренним устройством. А наличие альтернативных способов взаимодействия с устройствами на роботе позволяет использовать библиотеку без Rx.

Конечному пользователю предлагается работа с контроллером на разных уровнях абстракции, от средств для прямой манипуляции с I2C шиной, до готовых классов представляющих собой не отдельные функциональные компоненты, а целые модели, содержащие все необходимые устройства.

Все устройства на роботе, так или иначе, могут быть разделены по типу действия. Это либо сенсоры, которые только получают данные, но никак не могут влиять на окружающий мир, либо устройства, которые только выдают данные наружу. К первому типу относятся: гироскопы, акселерометры, аналоговые и цифровые датчики расстояний, датчики освещенности, температуры и давления. Ко второму – силовые двигатели и сервомоторы, световые ленты, лампочки.

Архитектура библиотеки очень гармонично укладывается в это разделение, навеянное ещё и устройством Observable-последовательностей. Каждый сенсор содержит метод ToObservable, вызов которого приводит к генерации событий из данных датчика. Классы, представляющие различные сенсоры, обеспечивают единообразное обращение с данными вне зависимости от внутреннего устройства датчика, корректное завершение работы и некоторые дополнительные функции. Силовые моторы, серво-приводы и светодиодные ленты напротив занимаются только приёмом информации в определенном формате и производят корректное выполнение полученных указаний.

4.1.1 Пара сенсор, устройство вывода

Общая практика при данном походе – использование F# pipeline operator (|>) для изменения данных полученных от сенсора с помощью функций из Obseravble модуля и Rx до того вида, который представляет вполне определенные команды для моторов, динамиков и различных других устройств вывода.

4.1.2 Пример

```
use model = new Model()
let rightMotor = model.Motor.["JM1"]
let leftMotor = model.Motor.["JM2"]
let speedFlow = model.AnalogSensor.["JA1"].ToObservable() |> Observable.map
(fun d -> if d > 55 then 100
          elif d < 45 then -100 else 0) |> Observable.DistinctUntilChanged()
use r_disp = speedFlow.Subscribe(rightMotor)
use l_disp = speedFlow.Subscribe(leftMotor)
```

Данный пример представляет собой реализацию простого робота с двухколесной базой, способного реагировать на препятствия. Аналоговый датчик расстояния (в примере AnalogSensor) выдает числа в диапозоне от 0 до 1024, эти показания зависят от освещенности и при приближении робота к препятствию нелинейно убывают. Экспериментально были получены числа 55 и 45, которые соответствуют 10 и 5 см.

Робот будет получать события из сенсора, последовательно применяя к ним функцию, которая переводит показания датчика расстояния в скорость которую нужно подавать на моторы. После преобразования мы получаем последовательность 100, 0 и -100 в зависимости от того как близко робот находится к стенке. Функция DistinctUntilChanged предотвращает лишние обновления в итоговой последовательности, в случае, если показатель скорости не менялся с прошлого раза. Такой подход позволяет заметно снизить нагрузку на I/O, предотвращая запись одних и тех же данных.

После того, как мы сконструировали новую последовательность (power), мы, буквально, подписываем наши моторы на эти изменения. Теперь робот будет ехать прямо, до тех пор пока не встретит на своем пути препятствие, тогда он остановится, если же препятствие после этого само начнет приближаться к нашему роботу, то он отъедет от него подальше.

4.1.3 Более сложная конфигурация

Роботы, как правило, представляют из себе более сложные конструкции, чем связка из одного сенсора и устройства вывода. А эффективное управление связано с одновременной обработкой данных сразу из нескольких сенсоров и выполнением сразу нескольких действий. Поэтому для удобства использования библиотека уже содержит класс, представляющий наиболее общую комплектацию модели со всеми возможными устройствами. Также при инициализации модели нет необходимости указывать конфигурацию устройств, в случае если они совпадают с стандартными устройствами из набора TRIK. Навигация по устройствам,

которые присутствуют в модели во множественном числе осуществляется по названиям соответствующим официальной документации TRIK-контроллер. Использование консолидированной сущности с готовыми параметрами упрощает код для простых моделей. Пример в начале заголовка уже использует Model для создания робота.

4.2 Работа с сенсорами

Важной частью устройств робота являются различные датчики. Именно благодаря ним робот способен осуществлять навигацию и понимать что-то об окружающей среде. Для своей работы робот должен постоянно обновлять данные из сенсоров. Кроме того, как мы уже видели, генераторы событий, а именно различные датчики, являются ключевыми точками в разработке реактивных программ, поэтому удобная работа с ними является основной задачей при разработке библиотеки.

Одним из самых важных требований является возможность не блокировать поток выполнения во время чтения данных из сенсоров. Датчики условно могут быть разделены на два типа по способу предоставления данных. Структурные различия этих устройств требуют разных методов решения.

- Polling sensors
- FIFO sensors

Polling сенсоры посылают данные несколько сотен раз в секунду. Обработка такого количества данных поглощала бы большую часть процессорного времени. И стремление получать и обрабатывать самые свежие данных в их полном объеме привело бы к увеличению времени отклика и зависанию. Более того, многие аналоговые датчики обладают характерными шумами, что ещё сильнее усложняет правильное управление. Ситуацию спасает то, что обычным роботам просто не нужно так быстро менять свое поведение, и получение данных раз в 50ms вполне допустимо.

FIFO – другой тип датчико, которые действуют как некий буфер, в который периодически поступают данные. Получения результата между двумя произвольными промежутками времени не гарантируется, а если гарантируется, то не зависит от пользователя библиотеки. Чтение данных из сенсора происходит совершенно прозрачно для программиста и файловой системы. Всё это может затянуть чтение файла на неопределенный срок. В месте с чтением ждать будет и программа, что никак не допустимо при разработке time-critical программного обеспечения. На время зависания робот будет не состоянии контролировать своё поведение, поэтому чтение таких сенсоров нужно переносить из главной ветки приложения. И в том и

в другом случае проблемы связанные с чтением датчиков могут быть решены с помощью использования Observable последовательностей, позволяющих активно работать с распараллеливанием.

4.3 Процесс разработки

F# входит в базовую поставку Visual Studio начиная с 2010, Linux и Mac есть полноценные IDE для разработки на .NET с плагинами для F#. Так что большим преимуществом при разработке роботов на базе платформы .NET является возможность выбирать не только IDE, но и ОС.

Роботы проекта TRIK снабжены Wi-Fi точной доступа и способны самостоятельно разворачивать сеть на своем борту. Работа с контроллером по сети и передача файлов на робота осуществляется с помощью ssh-туннеля и scp. В Linux это решается стандартными средствами, на windows можно использовать Putty и winSCP.

Mono на любой платформе работает с исполняемыми файлами, состоящими из managed блоков, полученными при компиляции на host-машине, так что при таком подходе отпадает необходимость в совершенно стандартной для разработки ПО на встроенных системах процедуре установки на host компьютер кросс-компиляторов и различных SDK, специфичных для каждого отдельного контроллера. Не говоря о том, что стандартной ОС для такого подхода является Linux, а попытки разработки на других ОС чреваты провалом.

Таким образом всё, что необходимо для разработки:

- Совершенно обычная IDE для разработки на .NET
- Средства для работы с сетью

Такая свобода выбора и отсутствие специфичного ПО позволяют понизить уровень вхождения и упросить работу для программистов, которые только начинают интересоваться робототехникой.

5 Системные задачи

При разработке библиотеки очень важно учитывать эффективность функций, которые будут доступны пользователям, поэтому большое внимание в работе было уделено оптимизации. Ограниченность ресурсов встроенной системы позволяет отчетливо наблюдать некоторые недостатки, невидимые при разработке приложений на ПК и серверных компьютерах. Прежде всего это связано с временными задержками при запуске программы, вызванными загрузкой и JIT-компиляцией используемых модулей, которые хранятся в отдельных *.dll. Использование больших исполнимых файлов чревато зависанием при старте на 30-40 секунд Несмотря на то, что после загрузки программа не подвержена зависаниям, такое поведения кажется неудовлетворительным. Полный или частичный отказ от использования библиотек написанных для .NET лишила бы задачу своей специфики. К счастью с помощью некоторых приемов можно уменьшить время загрузки почти втрое.

5.1 AOT

AOT или Ahead Of Time – вид компиляции применяющийся для языков с JIT. В отличие от JIT-компиляции производится до исполнения программы и, как следствие, не требует дополнительных ресурсов памяти во время исполнения. Кроме того, отсутствие требований на минимальное время работы компилятора (как в JIT) и возможность анализировать весь исходный текст программы позволяет применять более эффективные методы оптимизации.

Mono активно используется при разработке приложений на iOS и Android. Apple iOS не разрешает одновременное использование динамических библиотек и JIT компиляции, кроме того генерацию кода во время исполнения сложно назвать лучшим выбором для встроенных систем. Mono поддерживает AOT-компиляцию, при этом предусматривая полную (запрещающего JIT во всех модулях используемой программы, актуально для iOS), metadata-only, стандартную и максимально возможную (но с разрешением использования JIT) компиляцию. Кроме этого Mono предлагает несколько режимов оптимизации исполняемого кода. В качестве результата в папке с модулем появляется *.so файл, который будет использован Mono при следующем запуске.

Запуск Mono с различными режимами AOT-компиляции для файлов с библиотекой, исполняемых файлов библиотеки Rx, System.Fsharp.Core.dll, mscorlib.dll и ещё некоторых используемых в программе исполняемых файлов сокращает время загрузки в два раза

5.2 Tmpfs

Опять же для уменьшения времени загрузки исполняемых модулей с помощью стандартных средств ОС Linux в папку с приложением смонтирована tmpfs размером в 40 МБ, Файловая система расположеная прямо в оперативной памяти позволяет почти в полтора раза сократить время запуска.

5.3 Декомпиляция как одно из основных средств в разработке

Из-за большего различия языков C#, который является основным языком для разработки на .NET и объектно-ориентированной природы IL с языком F#, который берет свои корни в ML. F# имеет 3 довольно специфичную кодогенерацию, обеспечивающую полною совместимость этих языков. Поэтому при разработке следует контролировать, что представление данных и функций языка F# эффективно транслируются во внутреннее представление на CLI.

Небольшие изменения кода могут очень сильно повлиять на внутреннее представление, потому что код на F# будет переведен не в стандартные средства .NET, а в F# специфичное представление, требующее большего числа вызовов и преобразований. Разница которая может быть не заметна на настольных компьютерах может стать неприятной при программировании роботов. Поэтому после добавления новой функциональности в проект, .dll с библиотекой декомпилируется и проверяется на наличие нежелательных представлений. Для декомпиляции используется ILSpy и dotPeek

6 Применение

6.1 Использование библиотеки для обучения студентов кафедры

Кафедра Системного программирования много лет практикует обучение функциональному программированию студентов как закономерный этап после обучения их языкам C/C++ или Java.

Использование современных технологий и разработка на языках высокого уровня в программирования роботов имеет как минимум два больших преимущества:

1) Возможность создавать по-настоящему интерактивные робототехнические приложения, физически реагирующие с окружающим миром, и программы для встроенных систем без ограничения на выбор доступных средств и необходимости касаться низкоуровневого программирования в разработке.

2) Умелое применение средств, с которыми позволяют работать современное ПО, требует их глубокого понимания. Изучение новых парадигм программирования позволяет расширить кругозор решения задач новыми техниками, существующими не на уровне конкретного задания, а целой теории. При этом некоторые приемы могут сильно отличаться от всего, что вам приходилось до этого изучать. Лямбда-исчисление, замыкания, каррирование, событийно-ориентированное программирование, монадические выражения: асинхронные и ленивые вычисления поначалу могут вызывать некоторые сложности. Наглядность всегда была одним из основополагающих факторов в обучении и возможность знакомиться с этими мощными средствами с помощью такого показательного средства как роботы является большим плюсом.

7 Результаты

Реализована библиотека, которая поддерживает работу со всеми доступными устройствами на роботе (силовые и серво-моторы, кнопки, инфракрасные датчики расстояний, гироскопы, акселерометры). Основой упор сделан на использовании библиотеки с Rx и Observable последовательностями, но реализован и базовый интерфейс для работы с датчиками и моторами, который можно использовать как при выводе промежуточных результатов для отладки и тестирования, так и при разработке приложений без использования реактивной парадигмы.

Произведен анализ узких мест и выработан небольшой набор оптимизаций и рекомендаций для использования .NET на встроенных системах

Работоспособность проверена на примерах, написанных на разных языках (F#, C#)

Результаты работы были продемонстрированы на конференции СПИСОК 2014, библиотека использовалась в мастер-классах на конференциях Skolkovo Robotics 2014 и Microsoft DevCon.

Исходный код проекта доступен по адресу https://github.com/kashmervil/Trik-Observable

8 Список литературы

[1]Behavior based robotics Arkin, May 22, 1998 | ISBN-10: 0262011654

[2] Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

[3] Reactive Programming with Events, , MASTER THESIS, Tom?? Pet???ek, Charles University in Prague, Faculty of Mathematics and Physics
http://tomasp.net/academic/theses/events/events.pdf

[4]F# 3.0Expert programming Don Syme, October 31, 2012 | ISBN-10: 1430246502

[5] Offical Mono Documentation, http://docs.go-mono.com/

[6] Indroduction to Rx http://www.introtorx.com/

[7]Arrows, Robots and Reactive Functional Programming,
http://www.staff.science.uu.nl/ jeuri101/afp/afp4/hudak.pdf

[8] Learning Yampa and Functional Reactive Programming,
http://www.haskell.org/haskellwiki/Yampa