ПРИМЕНЕНИЕ DSL ДЛЯ ГЕНЕРАЦИИ ИСХОДНОГО КОДА ВСТРОЕННЫХ РЕАКТИВНЫХ СИСТЕМ

Толмач Евгений Иванович

PhD студент, Молдавский Государственный Университет Республика Молдова, г. Кишинёв

ON USING CUSTOM DSL FOR GENERATION OF EMBEDDED REACTIVE SYSTEMS

Evgheni Tolmaci

PhD student at Moldova State University Moldova, Chisinau

Аннотация. В последнее десятилетие Интернет Вещей неуклонно растёт. Глобальный рынок распределённых устройств нуждается в новых решениях, которые будут соответствовать современных трендам: платформо-независимость и высокая модульность. Процесс разработки встроенного ПО также нуждается в оптимизации.

Эта статья предлагает решение вышеизложенной проблемы. В ней представлен предметно-ориентированный язык для автоматической генерации реактивных систем. Его целью является генерация исходного кода для различных платформ, используя модули и конфигурационные файлы.

Abstract. Over the last decade the IoT (Internet of Things) is constantly accelerating its growth. The global market of distributed devices with embedded software on board is demanding the new solutions to meet the new trends: platform independency and high modularity. The R&D process itself is also demanding optimization.

This paper proposes a solution for the problems enumerated above. It presents a new Domain-Specific Language (DSL) for an automated generation of embedded actor-oriented systems. Its purpose is to generate the code for different platforms using the same modules and configuration files.

Keywords: actor, message broker, message-oriented middleware (MOM), process scheduler, domain specific language (DSL), actor-oriented software, embedded software, Internet of Things.

Вступление

Индустрия встроенного ПО относительно молода. Она расцвела с появлением персональных компьютеров и переносимых устройств. Прогнозируется, что глобальный рынок встроенного ПО вырастет с ~$10 млрд. в 2016 году до ~$19 млрд. в 2022. Ожидается, что среднегодовой темп роста будет на уровне 9 % [2]. В то же самое время Интернет Вещей делает свою революцию движимую относительно дешёвыми устройствами и повышающейся скорости беспроводного соединения. Рынок IoT на глобальном рынке, вероятно, вырастет до ~$11 трлн. [12].

Значительная доля ПО для IoT во многих аспектах зависит от платформы. Следовательно, для повторного использования уже реализованной логики, исходный код должен быть перекомпилирован или переконфигурирован в зависимости от встроенной платформы.
В некоторых случаях исходный код для различных платформ не совместим между собой, что в свою очередь может потребовать вмешательства для адаптации кода.

Целью данной статьи является оптимизация процесса адаптации при помощи DSL для автоматической генерации исходного кода под различные платформы.

1. Встроенное программное обеспечение на базе модели акторов

Встроенное ПО на базе модели акторов состоит из трёх основных компонентов: акторов, брокера сообщений и диспетчера задач.

Акторы являются инкапсулированными реактивными объектами, способными взаимодействовать друг с другом посредством передачи асинхронных сообщений. Эти вычислительные единицы в ответ на полученные сообщения могут выполнить следующее: отправить конечное число сообщений другим акторам, создать конечное число акторов, изменить внутреннее состояние, изменить своё поведение при получении других сообщений [3; 4]. Последовательность доставки и обработки сообщений не гарантируется, а некоторые действия могут выполняться параллельно [4]. Архитектура актор-ориентированных систем должна следовать философии "всё является актором".

Брокер сообщений является промежуточным ПО. Он является основным компонентом паттерна Message-Oriented Middleware (MOM). Главной задачей брокера является валидация, перевод и доставка сообщений между различными компонентами слабосвязанной системы. Он играет роль посредника в приложении, логически разделяя компоненты. Имплементация брокера сообщений базируется на одном из двух архитектурных паттернов: hub-and-spoke или сервисная шина. [5; 9].

Диспетчер задач это ещё один главный компонент ПО, построенного на базе модели акторов. Его целью является выполнение следующих задач [10]:

- Максимизация *пропускной способности*
- Минимизация *времени ожидания*
- Минимизация *времени выполнения*
- Максимизация *справедливости распределения ресурсов*
- Соблюдение *сроков выполнения*

Существуют четыре широко известные архитектуры планирования: простой Round Robin, Round Robin с прерываниями, Round Robin с прерываниями и функциональными очередями, Операционные Системы реального времени (RTOS). При разработке архитектуры встроенного ПО следует выбирать простейшее решение, соответствующее требованиям. Дополнительная сложность приводит к неоправданной потере времени при разработке и поддержке ПО. Также это может привести к падению производительности [1; 13].

2. Концепция

Акторы по определению являются атомарными вычислительными единицами (с точки зрения инкапсуляции). Они могут быть стандартизированы, модуляризированы и выделены в многоразовые библиотеки/
модули. В то же самое время, диспетчер задач и брокер сообщений также являются целями стандартизации и повторного использования кода. Таким образом имплементация реактивного ПО может быть сведена к определению головного контроллера, ответственного за следующие задачи:

- Импорт многоразовых библиотек/модулей
- Импорт многоразового брокера сообщений
- Импорт многоразового диспетчера задач
- Инициализация и конфигурация начальных акторов
- Инициализация и конфигурация брокера сообщений
- Инициализация и конфигурация диспетчера задач
- Регистрация акторов в брокере сообщений
- Планирование задач

На рисунке 2.1 представлен пример исходного кода головного контроллера встроенного ПО основанного на модели акторов. Имплементация может отличаться в зависимости от языка программирования и API применяемых библиотек, однако сама концепция остаётся неизменной.

```
// Import the libraries
#include <a_broker_library.h>
#include <an_actor_library.h>
#include <a_scheduler_library.h>

int main() {
    // Initialize the initial actors
    Actor ledActor = LedActor(PIN_1);
    Actor serialPortListenerActor = SerialPortListenerActor(9600);

    // Initialize the message broker
    MessageBroker broker = BufferedMessageBroker();
    broker.register("led", ledActor);
    broker.register("serial", serialPortListenerActor);

    // Initialize the scheduler &
    Scheduler scheduler = RoundRobinScheduler();
    scheduler.schedule(ledActor);
    scheduler.schedule(serialPortListenerActor);
    scheduler.run();
}
```

Рисунок 2.1.

Как мы можем заметить, головной контроллер, являясь входной точкой программы, выполняет лишь конфигурационную функцию. Следовательно, если мы унифицируем API всех акторов, брокера сообщений и диспетчера задач, то мы сможем генерировать исходный код головного контроллера автоматически на основе конфигурационных данных. Это позволит повторно использовать конфигурацию для генерации контроллеров для различных платформ и языков программирования.

3. Спецификация предметно-ориентированного языка (DSL)

Для того чтобы разделить конфигурацию от платформы/языка, необходим язык для чёткого и независимого описания конфигурации. Существует множество языков разметки, которые могли бы удовлетворить нашим условиям (XML, JSON, YAML, TOML и др.), но, несмотря на их преимущества, в некоторых случаях они могут привести к существенному потреблению памяти и процессорного времени для обработки. Другой проблемой является излишняя многословность [6]. С другой стороны они не предоставляют возможности внедрения переменных среды и встроенных выражений.

В качестве альтернативы существующим языкам разметки может быть разработан предметно-ориентированный язык (DSL). *DSL* – языки программирования, предназначенные для выполнения задач в определённой предметной области. Они позволяют описать задачи минимизируя семантический разрыв между алгоритмом и исходным кодом [7; 11].

Мы разработаем DSL для описания генерации платформо-независимого и модульного кода для встроенных реактивных систем. Далее мы будем называть его PACT (от англ. Powerful ACTors). Основными целями нашего языка являются: сокращение шаблонного кода, модульность сгенерированного кода, модульность и платформо-независимость конфигурационных файлов.

3.1. Синтаксис языка и внутреннее представление

Модульность генерированного кода может быть достигнута при помощи стандартного инструментария языков программирования. Повторно-используемый код может быть вынесен в модули/библиотеки, которые должны храниться в централизованных репозиториях [8]. Для конфигурации реактивных сущностей мы ввели ключевое слово ‘actor’ (Рисунок *3.1*). Оно имеет следующие аттрибуты: name (уникальный идентификатор), address (логический адрес, применяемый для доставки сообщений), module (внешняя библиотека), class/type (идентификатор класса, включая пространство имён).

```
define actor named 'test' at '/foo/bar/address'
    from 'periphery/Relay.h' as 'electricity::Relay'
```

Рисунок 3.1.

Следуя принципу DRY, мы решили предотвратить также и дублирование кода в конфигурационных файлах. Некоторые определения акторов могут частично повторяться. Модульность PACT файлов может быть достигнута при помощи прототипирования. Мы ввели новую сущность ‘prototype’, которая будет хранить общие атрибуты. Она будет содержать список (идентичный актору) атрибутов, которые могут быть унаследованы актором. (Рисунок *3.2*)

```
define prototype named 'test-prototype' at '/foo/bar/address'
    from 'periphery/Relay.h' as 'electricity::Relay'

define actor named 'test' by 'test-prototype'
```

Рисунок 3.2.

Другим способом достижения модульности конфигурации является наследование самих PACT файлов. Оно позволяет достичь более высокого уровня абстракции. Повторяющиеся части конфигурации могут быть выделены в отдельный файл, позволяя другим PACT файлам использовать их повторно. На рисунках *3.3* и *3.4* приводится пример наследования концигурации.

```
name 'test-parent-file'
group 'test-group'
version '1.0.0'

define prototype named 'test-prototype' at '/foo/bar/address'
    from 'periphery/Relay.h' as 'electricity::Relay'
```

Рисунок 3.3.

```
name 'test-child-file'
group 'test-group'
parent 'test-parent-file'

define actor named 'test-1' by 'test-prototype'
define actor named 'test-2' by 'test-prototype'
```

Рисунок 3.4.

3.2. Инструменты

Для имплементации нашего DSL необходимо выбрать инструменты для разбора и интерпретации. Существует множество языков, фреймворков и инструментов, которые имеют необходимый функционал: ANTLR, XText, JetBrains MPS, Groovy, parboiled (библиотека для Java/Scala) и др. Мы выбрали Groovy – Java-совместимый объектно-ориентированный язык выполняемый на платформе Java. Он поддерживает лямбды, многострочные литералы и встроенные выражения. Кроме того, Groovy поддерживает цепные вызовы методов с возможностью пропуска точек и скобок (Рисунок *3.5)*:

```
define(actor).named('test').at('/foo/bar/address')
    .from('periphery/Relay.h').as('electricity::Relay')
```

Рисунок 3.5.

Как мы видим, цепные вызовы методов в Groovy очень близки к естественному английскому языку. Человек, знакомый с предметной областью, может с лёгкостью читать и писать такой код. Однако есть несколько важных моментов, о которых следует помнить. Цепные вызовы должны соблюдать строгую очерёдность. Другой проблемой являются зарезервированные ключевые слова языка Groovy, которые не могут использоваться для в названиях методов. Это может привести к игре слов и сделать код менее естественным и затруднительным для чтения. Несмотря на вышеописанные проблемы язык Groovy является одним из лучших инструментов для разработки DSL. Мы используем его в качестве базы для языка PACT.

3.3. Генерация кода

Генерация кода является одной из сложнейших частей PACT DSL. Генератор должен учитывать целевую платформу и иметь достаточное представление об её архитектуре. В то же самое время модель должна быть независима от генератора. Самым простым решением является имплементация собственного генератора для каждой платформы. Все генераторы должны иметь общий интерфейс. Таким образом, мы можем выбрать необходимый нам генератор в самом интерпретаторе. Целевая платформа может быть передана через аргументы командной строки или явно через параметр в PACT файле (Рисунок *3.6*).

```
name 'test-ATMega-config'
version '1.0'
target 'ATMega8'
name 'parent-config'
group 'meteo-sensors'
version '1.0'
// ... some actor definitions
```

Рисунок 3.6.

Рисунок 3.7.

Однако явное указание целевой платформы конфликтует с идеей независимости PACT файлов. Так каждая платформа должна иметь собственную конфигурацию. Разрешением является наследование конфигурации, описанное в предыдущем разделе. Так каждая платформа может наследовать общий функционал (Рисунок *3.7*) и расширять/переопределять некоторые специфические части (Рисунки *3.8, 3.9*).

```
name 'test-ATMega-config'
parent 'parent-config'
target 'ATMega8'
```

Рисунок 3.8.

```
name 'test-ArduinoUno-config'
parent 'parent-config'
target 'ArduinoUno'
```

Рисунок 3.9.

ВЫВОД

В этой статье рассмотрены подходы и технологии, позволяющие генерировать реактивный встроенный код, основанный на базовой модели. Используя собственный DSL можно разделить архитектуру и логику от самих встроенных платформ, таких как ARM, AVR, PIC, Arduino и других. Естественно, что предметно-ориентированные языки неидеальны и имеют свои проблемы. Однако позитивный эффект значительно преобладает над незначительными неудобствами.

Индустрия встроенного ПО растёт большими темпами и нуждается в качественных изменениях. Разработка программного обеспечения для каждой платформы по-отдельности является устаревшей моделью. Переход к новым подходам неизбежен. Он сыграет важную роль в революции на рынке встроенного программного обеспечения.

Список литературы:

1. Cook J.A., Freudenberg J.S. Embedded Software Architecture. EECS. 2008.

2. Global Embedded Software Market Research Report-Forecast 2022. Available at: https://www.marketresearchfuture.com/reports/embedded-software-market-2103 (accessed 17 May 2018).

3. Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Doctoral Dissertation. MIT Press. 1986.

4. Hewitt C. Viewing Control Structures as Patterns of Passing Messages. Journal of Artificial Intelligence. 1977. V. 8. No. 3. P. 323-364.

5. Kale V. Integration Technologies. Guide to Cloud Computing for Business and Technology Managers: From Distributed Computing to Cloudware Applications. CRC Press. 2014. ISBN 9781482219227.

6. Megginson D. Imperfect XML: Rants, Raves, Tips, and Tricks ... from an Insider. Addison-Wesley Professional. 2004. ISBN 0131453491. 256 p.

7. Mernik M., Heering J., Sloane A.M. When and how to develop domain-specific languages. ACM Computing Surveys (CSUR). 2005. No. 37 (4). P. 316-344.

8. Raghav G., Gopalswamy S., Radhakrishnan K., Delange J., Hugues J. Architecture Driven Generation of Distributed Embedded Software from Functional Models. Ground Vehicle Systems Engineering and Technology Symposium (GVSETS). 2009.

9. Samtani G., Sadhwani D. Integration Brokers and Web Services. Web Services Business Strategies and Architectures. Apress. 2013. ISBN 9781430253563.

10. Silberschatz A., Galvin P.B., Gagne G. Operating System Concepts (9 ed.). Wiley Publishing. 2012. ISBN 0470128720.

11. Spinellis D. Notable design patterns for domain specific languages. Journal of Systems and Software. 2001. P. ?

12. Stralin T., Ghanasambandam C., Anden P., Comella-Dorda S., Burkacky O. Software development handbook. Tranforming for the digital age. Software Development. 2016. McKinsey & Company, Inc. P. ?

13. Tyree J., Akerman A. Architecture Decisions: Demystifying Architecture. IEEE Software, IEEE Computer Society Press Los Alamitos. 2005. V. 22. P. 19-27. ISSN: 0740-7459.