STUDIA UNIVERSITATIS MOLDAVIAE, 2018, nr.2(112)
Seria "Științe exacte și economice" ISSN 1857-2073 ISSN online 2345-1033 p.26-30

CZU: 517.9:519.6

CODE GENERATION DSL FOR EMBEDDED ACTOR-ORIENTED SYSTEMS

Evgheni TOLMACI

Universitatea de Stat din Moldova

Over the last decades embedded software is becoming more complex and more distributed. Fuelled by revolution of Internet of Things (IoT) the global embedded market is constantly growing and is predicted to accelerate its growth. At the same time the embedded software is demanding the new solutions to still open problems, such as higher modularization, platform independency and optimized research and development (R&D) process.

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. It achieves the main goal by decoupling the modules, the configuration files and the platform from each other.

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

DSL PENTRU GENERAREA CODULUI ACTOR-ORIENTAT PENTRU SISTEME ÎNCORPORATE

În ultimele decenii software-ul încorporat devine mai complex și mai distribuit. Datorită revoluției internetului obiectelor (IoT), piața globală este în continuă creștere și se preconizează că creșterea economică va accelera în viitor. În același timp, software-ul încorporat are nevoie de soluții noi pentru rezolvarea problemelor deschise: modularizarea mai înaltă, independența de platformă și optimizarea procesului de R&D.

În acest articol se propune o soluție pentru rezolvarea problemelor enumerate mai sus. El prezintă un nou limbaj de programare (DSL) pentru generarea software-ului încorporat actor-orientat. Scopul lui este generarea codului pentru diferite platforme utilizând aceleași module și fișiere de configurare. Această lucrare realizează obiectivul principal prin decuplarea modulelor, fișierelor de configurare și platformelor încorporate.

Cuvinte-cheie*: actor, broker de mesaje, software integrator mesaj-orientat, planificator de procese, limbaj specific domeniului, software-ul actor-orientat, software-ul încorporat, Internetul Obiectelor.*

Introduction

The embedded software industry is relatively young. It emerged with the appearance of the computers and personal portable gadgets. The global market of embedded software is expected to grow from US ~$10 Billion in 2016 to US~$ 19 Billion by 2022. The average compound annual growth rate is expected to be approximately 9% [1]. At the same time Internet of Things (IoT) makes its own revolution driven by relatively cheap hardware and increasing connectivity. The global impact of the IoT as a part of digital landscape is predicted to reach ~$11 trillion [2].

A significant share of IoT is driven by embedded software which in many cases is platform-specific. So in order to reuse the already implemented logic, the source code needs to be recompiled or reconfigured depending on embedded platform. In some other cases the source codes for different platforms are not even compatible, so they may require deeper intervention in order to adjust and customize the code.

The goal of this article is to optimize the customization process by the creation of a domain specific language (DSL) for generating the platform-specific code.

1. Embedded Actor Oriented Software

The embedded actor-oriented software consists of three main components: actors, a message broker (one or many) and a process scheduler.

Actors are well encapsulated active objects which can communicate to each other by sending asynchronous messages. These computational entities in response to messages they receive, can do concurrently the following actions: send finite number of messages to other actors, create finite number of new actors, change their internal state, change their future behaviour on receiving other messages [3,4]. The sequence of message delivery and message handling is not guaranteed and some actions could be performed in parallel [3]. So the architecture of an actor-oriented system should assume these concepts and adopt the "everything is an actor" philosophy.

STUDIA UNIVERSITATIS MOLDAVIAE, 2018, nr.2(112)
Seria "Științe exacte și economice" ISSN 1857-2073 ISSN online 2345-1033 p.26-30

A *message broker* is an intermediary software module. It is a key component of message-oriented middleware pattern (MOM). The main responsibilities of a message broker are validation, translation and routing of messages between different components of a loosely coupled system. It acts as a mediator among applications, decoupling the components from each other. The implementation of message brokers is based on one of two architectural patterns: hub-and-spoke or message bus [5,6].

Process scheduler is another main component of an actor based embedded software. It aims to achieve some of the goals [7]:

- Maximizing *throughput* (the amount of processes executed per time unit)
- Minimizing *waiting time* (the time a process needs to wait once it has been created until it starts being processing)
- Minimizing *latency* (the time a process needs for completion starting from creation time)
- Maximizing *fairness* (fair CPU sharing according to the priorities of the processes)
- Meeting the *deadlines* (has very high importance for embedded systems)

There are four well known and widely used scheduling architectures (sorted in order of increasing complexity): Simple Round Robin, Round Robin with Interrupts, Round Robin with Interrupts and Function Queues, Real-time Operating System (RTOS). On the one hand the rule of thumb is to choose the simplest one that meets the requirements. Extra complexity will lead to undesirable waste of time during development and maintenance. On the other hand it may cause performance issues [8,9].

2. The Concept

The actors by definition are atomic computation units (from encapsulation point of view). They can be easily standardized, modularized and extracted into reusable libraries/modules. At the same time the process schedulers and the message brokers are another target of standardization, being highly reusable. Thus the implementation of actor-oriented software can be reduced to definition of a custom main controller which will be responsible for the following logic:

- Import of reusable actor libraries/modules
- Import of a reusable message broker library/ modules
- Import of a reusable process scheduler library/ modules
- Initialization and configuration of initial actors
- Initialization and configuration of the message broker
- Initialization and configuration of the process scheduler
- Registration of actors on the message broker
- Actor scheduling

Listing 2.1 presents an example of a main controller of an actor-oriented embedded system. The implementation may differ depending on language and APIs of the libraries used, but the main concepts will remain the same.

```
// 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();
}
```

Listing 2.1

As we can see the main controller has a configurational function only. So if we unify the APIs of all the actor classes and do the same for the message brokers then we will be able to generate the main controller based on configuration data. This would allow us to use a single configuration file for the generation of the main controllers for different platforms/languages.

3. DSL Specification

As we want to decouple the platform/language specific stuff from the configuration data, we need a language to express configuration in a clear and independent way. There are different mark-up languages meeting most of our requirements (such as XML, JSON, YAML, TOML and others), but despite their advantages, they do sometimes cause a significant increase in data size and processing time. Another problem is that some of them are too generic and too verbose [10]. also they do not provide out of the box a possibility to inject some environment variables (like execution time) and inline expressions.

As an alternative to existing generic mark-up languages a domain specific language could be developed. The *domain-specific languages (DSLs)* are the programming languages tailored to particular application domains. They allow the concise description of an application's logic reducing the semantic distance between the problem and the program [11,12].

We will design a DSL for the definition of a cross-platform and modularized code generation for embedded actor-oriented systems. Let's call it PACT (Powerful ACTors). The main goals of the new language are: elimination of hand-written boilerplate code, high modularity of generated code, high modularity of configuration files and cross-platform code generation (for different platforms using the same configuration file).

3.1 Internal Model and Language Syntax

The modularity of generated code can be achieved by standard modularization tools of embedded programming languages. Thus the reusable code can be stored in modules (e.g. C++ libraries) instead of placing directly in the main controller. The libraries are developed over time and should be collected in a centralized repository [13]. At the same time we introduce in our PACT language a new entity marked with keyword 'actor' (*Listing 3.1*). It will have the following attributes: name (a unique identifier), address (used for message routing by message broker), module (external library), class/type (the name of the class including its namespace).

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

Listing 3.1

According to the DRY principle we want to eliminate the duplication of code in PACT files. The modularization of the PACT file itself can be achieved by prototyping. Some actors defined in our configuration file may partially repeat the configuration of the other ones. This can be elegantly solved by introducing an entity called 'prototype', which will be used for storing the common attributes. It will have the same set of attributes as 'actor' has. At the same time we add a new attribute 'prototype' to the 'actor' entity. In this case an actor definition can easily inherit the common configuration attributes from a prototype. (*Listing 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'
```

Listing 3.2

Another way to achieve high modularity of PACT configuration files is to introduce inheritance of PACT files. It will help to achieve a higher level of abstractization. The common configuration stuff can be shared among different files allowing extraction of potentially repeatable parts of configuration to the parent level. The *listings 3.3* and *3.4* demonstrate an example of configuration inheritance.

```
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'
```

Listing 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'
```

Listing 3.4

3.2 Tools

To implement our own DSL we need to choose a tool for parsing & interpreting. There are multiple languages/frameworks/tools providing such functionality: ANTLR, XText, JetBrains MPS, Groovy, parboiled (Java/Scala library), etc. Our choice is Groovy. It is a Java-syntax-compatible object-oriented programming language for the Java platform. It supports closures, multiline strings and expressions embedded in strings. Moreover Groovy supports chained methods with possibility to avoid periods and parentheses. Thus the DSL statement from *Listing 3.1* can be interpreted as a plain Groovy code (*Listing 3.5)*:

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

Listing 3.5

As we can see the chained Groovy command expressions are very close to natural English language. A human familiar with the domain can read and write this code with ease. However there are a couple of problems to be wary of. The chained expressions should follow the strict order to be valid. Another pitfall is that there are some Groovy grammar keywords which should be used carefully or should not appear in the DSL at all. This may sometimes lead to the need of word-twisting and thus making the code look less natural and harder to read. Despite these pitfalls Groovy is almost a perfect fit for the development of a DSL. We will use it as a base for development and implementation of the PACT language.

3.3 Code generation

Code generation is one of the trickiest parts of PACT DSL. It should be aware of the target platform and have sufficient knowledge about its architecture. But the problem is that we want to decouple the model from the generator. The easiest solution is to create a custom implementation of common generator interface for each target platform. Thus we will be able to choose the generator we need. The target platform can be provided as a parameter to the main controller of PACT interpreter. Another option is to introduce a new keyword in our DSL called 'target' in order to specify explicitly the target platform (*Listing 3.6*).

```
name 'test-ATMega-config'
version '1.0'
target 'ATMega8'
```

Listing 3.6

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

Listing 3.7

At the same time any explicit specification of the target platform will break the idea of loosely coupled code generation. Each target platform will need its own copy of PACT configuration. However this problem can be simply worked around by using inheritance as explained in previous section. Thereby the platforms can inherit the common configuration (*Listing 3.7*) and extend/redefine some specific parts (*Listings 3.8*, *3.9*).

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

Listing 3.8

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

Listing 3.9

Conclusion

In this paper we discussed the approaches and technologies which allow generation of actor-oriented embedded software based on an initial model. Using a custom DSL the architecture and business logic can be easily decoupled from custom embedded platforms, such as ARM, AVR, PIC, Arduino and others. Of course the DSL approach is not ideal and has some pitfalls. But the positive effect of this approach has a significant advantage over the inconveniences caused by those corner cases.

The embedded software industry is growing fast and in order to meet the demands of the market it needs to make a qualitative leap. The development of embedded systems for each platform separately is an outdated approach. So switching to modern approaches (such as the one described in this paper) is inescapable and will play a key role in revolutionizing the market and building the future.

References:

1. *Global Embedded Software Market Research Report – Forecast 2022.* https://www.marketresearchfuture.com/reports/embedded-software-market-2103 (Retrieved on 17 May 2018)
2. STRALIN, T. GHANASAMBANDAM, C., ANDEN, P., COMELLA-DORDA, S., BURKACKY, O. *Software development handbook. Tranforming for the digital age*, 2016. Software Development. McKinsey & Company, Inc.
3. HEWITT, C. Viewing Control Structures as Patterns of Passing Messages. In: *Journal of Artificial Intelligence*, 1977.
4. GUL, Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*, 1986. Doctoral Dissertation. MIT Press.
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. SAMTANI, G., SADHWANI, D. *Integration Brokers and Web Services*. Web Services Business Strategies and Architectures, 2013. Apress. ISBN 9781430253563
7. SILBERSCHATZ, A., GALVIN, P.B., GAGNE, G. *Operating System Concepts* (9 ed.). Wiley Publishing, 2012. ISBN 0470128720
8. COOK, J.A., FREUDENBERG, J.S. *Embedded Software Architecture*. EECS, 2008.
9. TYREE, J., AKERMAN, A. *Architecture Decisions: Demystifying Architecture*. IEEE Software, IEEE Computer Society Press Los Alamitos, 2005. ISSN: 0740-7459
10. MEGGINSON, D. *Imperfect XML: Rants, Raves, Tips, and Tricks ... from an Insider.* Addison-Wesley Professional, 2004. ISBN 0131453491
11. MERNIK, M., HEERING, J., SLOANE, A.M. *When and how to develop domain-specific languages*. ACM Computing Surveys (CSUR), 2005.
12. SPINELLIS, D. Notable design patterns for domain specific languages. In: *Journal of Systems and Software*, 2001.
13. 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.

Date despre autor:
Evgheni TOLMACI, doctorand, Școala doctorală *Matematică și Ştiinţa Informaţiei*, Universitatea de Stat din Moldova.
E-mail: evgeniy.tolmach@gmail.com

Prezentat la 02.07.2018