EventGateways: Combining Escala Events with Actors

Jurgen M. Van Ham
Software Technology Group
Technische Universität Darmstadt

ASCOLA group; EMN-INRIA, LINA
Dépt. Informatique. École des Mines de Nantes
jurgen.van-ham@mines-nantes.fr

ABSTRACT

Event-Based programming has been successfully combined with concurrency, this was studied in languages like Event-Java. There is research on extending Aspect-Oriented Programming with concurrency. The relation between Event-Based Programming with Aspect-Oriented programming has be studied by several languages. Combining both styles with concurrency in a single language is possible with the right building blocks. After using the join operator as a building block, we study an alternative, that is inspired by the actor model. This can help programmers to make a step towards concurrency.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language constructs and Features—*Concurrent programming structures*

General Terms

Languages, Experimentation

Keywords

concurrency, declarative events, actors, aop

1. PROBLEM AND MOTIVATION

Before talking about concurrency, we discuss the goals of both the Aspect-Oriented Programming [14] (AOP) and the Event-Based Programming (EBP) paradigms and how concurrency was already studied with both of these. We show that Aspect-Oriented Programming can benefit for extending an event system, because of the relationship between these paradigms, which was studied in some languages that use an event system to offer aspect-oriented programming.

For at least twenty centuries, people have been using divide-and-conquer as a way to deal with problems in general. When this is applied to software there are several approaches to create programs by organizing the code in a structured way. One of the first important steps is structured programming [5]. Even when not all later steps are always unanimously appreciated, we consider Object-Oriented Programming [19] (OOP) an important step to a design where functionality of a program is represented by identifiable parts in the source code. The use of design patterns [10] helped to make object-oriented programs modular. However, this paradigm still does not allow to express every concern in a specific part of code.

Among the popular design patterns is the observer [10] also known as publish subscribe. This results in a tight coupling between the publisher and the subscriber of an event. An example of this coupling would be that we like to change the type that is published, this implies we need to change the subscriber as well. When a subscriber observes several publishers, such a change cause several other changes. *Event-Based Programming* offers support for the publish subscribe mechanism as part of the language. In this paradigm the publisher is called an event. The observers of the event are *handlers*, which can be registered with an event. We use the terminology from EScala [11] for EBP, the word event for the static entity in the source code, which can be declared like another variable or triggered. When an event is triggered during run time this becomes an event *occurrence*.

Apart from these limitations of the observer pattern, there are what is known as *crosscutting* concerns. These are the target of *Aspect-Oriented Programming* [14] (AOP). For instance, when we implement logging, which is one of the typical examples, in an OOP program there is no single component that is responsible for logging, since the statements for this logging will be spread over the program. AOP uses the image of a program as a piece of fabric and such concerns are colored threads that need to be woven into that fabric. The original program can be considered as threads of a single color in this imaginary carpet. After an aspect has been woven into a base program the result can be considered as a base program.

Some languages, for example EJava [16] and EScala [11], study the relationship between Aspect-Oriented Programming and Event-Based Programming. We use EScala for our experimental implementation that we describe in the next section.

Aspect-Oriented Programming can be seen a description of modifying a *base* program and the right places where these each modification has the desired effect. In the terminology of this paradigm, we say each *advice* needs to be inserted at an associated *pointcut* of the *base* program.

The initial view on Aspect-Oriented Programming is inserting pieces of code into a *base* program. This pieces of program are called an *advice*. Each advice is associated with the place where it is inserted. The place is called a *pointcut*, to specify the pointcuts, the aspect languages offer a pointcut language which can be considered as a domain speciic

SRC'13 ACM Student Research Competition Grand Finals.
ACM 0-89791-88-6/97/05.

language to select from the set of all places in the base program where code can be inserted those where a specific advice will be inserted. Once an aspect is woven into a base program, the result can be the base program for another aspect.

When using EBP to implement AOP, the pointcuts are implemented as events that are implicitly triggered by the execution of the base program reaches certain points in the code. An advice is can be considered as a handler associated with such an implicitly triggered event.

Many programs have concurrent behavior, this is not only for technical reasons like the evolution to multi-core processors. A functional reason for concurrency can be observed in programs that interact with users or other programs, we do not expect they stop all activity while waiting for input during this interaction.

We extend an event system with concurrency, because of the relation between AOP and EBP this concurrency can be studied in a Aspect-Oriented language. This might not fit with the image of inserting pieces of advice at places in the base program that are indicated by pointcuts, but concurrent aspects have been studied before as Concurrent Event-Based AOP [6] (CEAOP).

Our aim is to combine concurrency with Event-Based Programming. This combination we use to implement concurrent aspects. We want to make the introduction of concurrency for the programmer

2. BACKGROUND AND RELATED WORK

In concurrent programs the programmer is responsible for controlling the access to shared data. During a long time people study the way to write concurrent programs. The first language with support for concurrency is Concurrent Pascal [2]. With Java support for concurrency became available in a language that has been widely spread. However this did not convince a big amount people to start writing concurrent programmers. Most application programmers avoid to touch concurrency themselves, in general frameworks shield for most programmers for all exposure to concurrency.

An alternative where the programmer does not need to control access to shared data at all is the actor model [13], this is possible because in the pure model there exist no data that is shared between actors, which are concurrent entities. This model is used in the language Erlang [1], which is successful in the telecommunication industry. While the actor model did not prove to be best concurrency model, it offers an elegant programming style. It shields the programmer from some of the problems we associate with concurrency, mainly identifying and protecting critical sections.

Object-Oriented Languages can offer the actor model, this is not always the pure model. For example Scala offers a library to implement actors, but the arguments of the message are not copied, a way for the programmer to avoid problems with sharing data is to use only immutable data in messages. The language does not enforce this, therefore the programmer has to avoid any changes to this data. Since the message is not copied when sent to a mailbox the programmer can not change any part of the message even after sending it.

We use Event-Based Programming as a tool that is also useful for Aspect-Oriented Programming. Complex Event Processing [15] (CEP) system, like for instance EventJava [7], combine concurrency with events. These CEP systems are often middleware, which deals with events that are triggered outside of the system. In EScala the events are triggered inside a single sequential program, the events are handled always synchronous, because the single thread will stop executing a sequence of instruction when an event is triggered until all handlers of the event occurrence are executed. Similar to CEP systems EScala event occurrences can lead to other event occurrences, since performance is not as important for EScala as in CEP systems, transforming and filtering can be more complex, since the event system is not designed to achieve the highest number of events processed during a short time. The event system of EScala deals with internally triggered events, when the program slows down because the processing of events, the generation will happen later, the synchronous handling also will avoid losing any event occurrence because of congestion, unlike the CEP systems which cannot let incoming events to pile up.

The event system of EScala is sequential, it is possible to use the explicit concurrency of the underlying Scala [17] language, but the event system does not support this. We experimented before [20] with the integration of elements of the join calculus[9], which results into a programming style that is very different from sequential EScala once concurrency is used, because a programmer needs to learn to program with joins.

One of the differences mentioned in the survey by Eugster et al. [8] is between messages and invocation. The actors and the CEP systems use messages that are represented as data, an advantage of this representation is that event occurrences can be immediately stored in a buffer or a queue. EScala uses invocations for its event occurrences, the advantage for this that handlers can be called without any conversion. It is possible to convert event occurrences between messages and invocations. A conversion in both ways is only possible when the arguments of an invocation can be serialized.

For our experiments we limit the set of events to those that are triggered implicitly or explicitly in a program. We do not consider input from outside the system at all. Distributed programs can rely on external triggered events, but we do not involve these in our current experiment.

The event expressions of EScala can not deal with concurrent events like a CEP system does. We can extend this language with the ideas from the actor model. The event occurrences caused by triggering an event in EScala are not the same as messages in the actor model, but this difference we solve by a library.

3. APPROACH AND UNIQUENESS

We start from the idea of an actor, a single threaded entity that receives messages via a single mailbox, in our model an actor can be more than a single object. We can consider a group of objects as islands of sequential EScala code. The objects on this island can only indirectly observe those events that are not declared on the same island. The only entrance to such an island without any internal concurrency is an `EventGateway` instance, each island has a unique event gateway that represents it to other islands, this can be compared with an actor which has a single mailbox. Informally we use the concept event gateway not only for this object but also for the single threaded island itself, since it is represented by its `EventGateway` instance to the outside.

Registering a handler with an event that is triggered outside the event barrier leads to a second thread inside our

single threaded island. Because the thread that triggers the event on another island uses its own event to execute handlers that are part of our island, that also has its own thread. Similar to the actor model, where two threads inside an actor abandons the safety that the actor model provides. Therefore, we define an event on the island that is triggered after an observed external event is triggered, this internal *proxy event* represents the external event. When handlers are registered with this *proxy event* there is no thread that enters into the concurrency free island.

The remaining problem is that this event has to be triggered with the same argument as the external event that it represents. For this we will use the mailbox of the actor that is used inside an `EventGateway` object. Our library registers a handler to the observed external event that sends a message to the actor inside the event gateway. This message contains information about the triggering event and the parameter that was used to trigger the external event. When the actor reads this message from its mailbox, it triggers the proxy event.

The proxy event is triggered after the external event was triggered, this means the observed external event does not wait for the execution of handlers of its proxy event. This results into asynchronous triggering of the proxy events and their registered handlers.

From the implementation side this `EventGateway` contains a Scala actor, but at compile time the actors define on what types of messages they will react. To solve this we use for each event that is observed via an event gateway a unique number. The event gateway contains a mapping to trigger the representing proxy event for that number. The delayed events are also used to represent proxy events. For delayed events the event gateway will not specify an event triggered outside the island to observe via the mailbox of the internal actor. Starting the actor takes place when the first event occurrence will be sent to the actor. The handlers registered by the event gateway to the external events are responsible for transforming an invocation to a message with an unique id to the internal actor.

Each event gateway exposes a `startEvent` and `haltEvent` which can be observed by any handler on the island associated with that event gateway.

Inside an EventGateway island we can use almost the same programming style as a normal EScala program, except that we need to return the thread to the actor inside the `EventGateway` object, since this actor needs this thread to trigger the proxy events.

Therefore, the programmer has to design his program as a set of handlers that trigger before finishing an event to request the execution their successor. This requesting event needs to use the same (single) mailbox that is used for the proxy events. We can create these events as *proxy events* of an event that is defined inside the island. Instead of each time declaring an event and its associated proxy event we combine these two events into in a *delayed event*. As the name suggests, these events are not handled immediately after they are triggered. Since other events that received via the same the mailbox of the actor inside t he `EventGateway` object can be executed before. The actor model does not specify and order, but the implementation that we use internally implements a FIFO buffer.

There are other models like s like SOM [4], ProActive [3] that use the ideas from the actor model into an object language. But these languages do not have declarative events as EScala uses to offer Aspect-Oriented Programming.

4. RESULTS AND CONTRIBUTIONS

Inspired by the actor model as an elegant solution to concurrent programming offered, we combine sequential parts of EScala code to create a concurrent program. We build on top of Scala Actors [12], which are not pure actors as they rely on the programmer to deal with messages as if they are immutable, even when the library implementation does not enforce this. Therefore, we need to respect this rule also for any event occurrence that is received by an `event gateway` instance.

We study how the actor model can be integrated into a language that uses a powerful event system to offer next to Event-Based Programming also Aspect-Oriented Programming by offering events that implicitly triggered when the program executes a method during run time.

We build on top of a language that offers only synchronous events. Our experiment introduces asynchronously handled events in case the event occurrences are received via an *event gateway* construct. The events that are triggered outside the event gateway are always triggered via the event gateway. For events triggered inside the *delayed events* offer the same asynchronous behavior.

An experiment shows that splitting a sequential EScala program into concurrent parts is still not solved by our actor inspired approach. We used for this the simulation of a world with animals and plants that was used for the evaluation of EScala [11]. A first issue when introducing concurrency into a sequential event based program is that a programmer needs to detect any method call that interacts with objects from another island. These method calls need to be converted to use the associated event gateway of the island, without any tool the programmer probably forget one or more. Another observation is that in a program which is designed for a sequential environment, the programmer makes assumptions on atomic actions that are no longer valid in concurrent environment.

This kind of problems indicate that our actor-based model does not help to transform any sequential program into a concurrent program. We wrote experimental programs with this that are successful, but these we wrote from scratch, this avoids any wrong assumptions on atomic behavior of any action. The most simple one is a simulation of the celestial system of sun, earth and moon that detects a full moon situation. As a next step we implemented a concurrent version of the sensor and camera running example from [18], here the sensor and the camera interact but they can act concurrently.

These experiments show a programming style that is similar to programs in the original sequential EScala programs. On a single threaded island can be proxy events and delayed events which are created via method call to the event gateway of that island. The rest of the code can be similar. Because there is a only a single thread available on each island the code on this island needs to return this thread to the event gateway, this allows the event gateway to handle event occurrences received from outside of the island.

As our original goal was to find an alternative for the integration of join calculus elements into EScala [20], we see that the actor inspired model does not offer trivial support to make the execution of a method wait for another event.

This actor model offers in some cases a simple model but when using this model we also discovered some disadvantages.

However, when using the elements from join calculus, making the execution of a method wait for another event is very straight forward when we use an implicit event that is triggered before execution that method and using this event as input for a binary join. To implement a similar mechanism with the actor inspired model will be more complex because each event that is observed by an island in another gateway has an asynchronous behavior.

This actor inspired model leads sometimes to easier programs, but not all program are fit for this model. But it is not yet clear how we can know this before implementing that program. One indication that a program will not fit, is when we like to block the execution before or after a method, the asynchronous nature does introduce extra complexity for the programmer. We believe that the actor inspired model can be useful for a specific group of problems, but identifying which problem belong to this group would need further studying. We hoped to to use this actor-based model as a tool to extend Aspect-Oriented programming. But we consider the missing immediate support for synchronous events, that can block the triggering thread when desired, at this moment a motivation to study other approaches.

Therefore, we consider now to integrate more elements of join calculus into the event model of EScala. For the programmer, this join based alternative might require some extra time to learn using it, but we believe the model will be more powerful than the actor-based model we studied here.

5. REFERENCES

[1] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, 2010.

[2] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1,2:199–206, June 1975.

[3] D. Caromel. ProActive Parallel Suite: Multi-cores to Clouds to autonomicity. In *Intelligent Computer Communication and Processing, 2009. ICCP 2009. IEEE 5th International Conference on*, 2009.

[4] D. Caromel, Luis Mateu, and É. Tanter. Sequential Object Monitors. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, pages 316–340, 2004.

[5] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11:147–148, March 1968.

[6] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 79–88, New York, NY, USA, 2006. ACM.

[7] Patrick Eugster. EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, pages 570–594, 2009.

[8] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *Computing Surveys (CSUR*, 35(2):114–131, June 2003.

[9] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, USA, 1996. INRIA Rocquencourt.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] Vaidas Gasiūnas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: Modular event-driven object interactions in Scala. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011*, Porto de Galinhas, Pernambuco, Brazil, March 2011. ACM Press.

[12] Philipp Haller and Frank Sommers. *Actors in Scala*. Artima Inc, January 2012.

[13] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming - 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finnland, June 1997. Springer-Verlag.

[15] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[16] Angel Núñez. *Une modèle de programmation pour la intégration de classes, événements et aspects*. PhD thesis, Université de Nantes and École des Mines de Nantes, June 2011.

[17] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[18] Hridesh Rajan and Kevin J Sullivan. Unifying Aspect- and Object-Oriented Design. *ACM Transactions on Software Engineering and Methodology*, 19(1), April 2008.

[19] Tim Rentsch. Object oriented programming. *SIGPLAN Not.*, 17:51–57, September 1982.

[20] Jurgen M. Van Ham. Adding high-level concurrency to EScala. In *AOSD Companion '12: Proceedings of the 11th annual international conference on Aspect-oriented Software Development Companion*. ACM, March 2012.