On Liberating Programs from the von Neumann Architecture via Event-Based Modularization

Somayeh Malakuti *
Software Technology group
Technical University of Dresden, Germany
somayeh.malakuti@tu-dresden.de

Mehmet Aksit
Software Technology group
University of Twente, the Netherlands
m.aksit@utwente.nl

Abstract

From the early days of computers, researchers have been trying to invent effective and efficient means for expressing software systems through the introduction of new programming languages. In the early days, due to the limitations of the technology, the abstractions of the programming languages were conceptually close to the abstractions of the von Neumann based realization platforms. With the advancement of the technology, computers have been increasingly applied for complex problems in different application domains. This required the challenge of designing programming languages that resemble more the semantics of software rather than the concepts of underlying machinery. To this aim, various new language concepts, such as object-oriented, aspect-oriented, and event-based languages have been introduced. While these languages were successful in enhancing the expression power of languages towards more semantic concerns of application domains, they fail in short in representing emergent behavioral patterns of software effectively. We outline a set of requirements to overcome these shortcomings, and explain the concept of **event-based modularization** as a possible solution.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Languages, Design

Keywords Emergent behavior, modularity, von Neumann architecture

1. Introduction

Although there are many facets of software engineering, its main objective is to create software systems that execute on hardware/-software platforms [3, p.9-12]. From the early days of computers, researchers have been trying to invent effective and efficient means for expressing software systems through the introduction of new programming languages.

Along this line, we observe a continuous interplay between programming languages and platforms. We claim that to create better languages, one needs to understand the historical nature of this interplay and the forces that influence it. From this perspective, we will make an attempt to characterize platforms, programming languages and the constraints among these.

During the last decades platforms have evolved dramatically. They are now many magnitudes more powerful than their previous generations, they consist of multicores and they are based on geographically distributed networked architectures. Despite these developments, at their core, platforms still display typical characteristics of von Neumann architecture of instructions, registers and memory locations. In the literature [1, 5], programming languages are generally classified based on the paradigms that they adopt. Functional, logic, and imperative languages are considered as the three basic categories; they all have equivalent expression power in the sense that any executable program can be expressed in one of these categories.

In practice, however, the main reason why a programming language might be selected is more based on the pragmatic factors such as availability of the tools, programmers' skills and on the desired non-functional characteristics of programs such as adaptability, flexibility and evolvability rather than the paradigm of the language. Moreover, most commercial languages are not based on a single paradigm but they borrow features from multiple paradigms. Imperative languages have been much more successful in practice and therefore in this paper we will mainly focus on these languages.

Every language provides a set of first-class abstractions that are directly supported by the mechanisms of that language. For example, a first-class abstraction may be passed as an argument of a call, it may be returned as a result of a call, it may be stored or retrieved, etc. The interplay between languages and platforms is defined by the following constraint [1]: the first-class abstractions of a language as much as possible must match natural abstractions and semantics of the considered problem domain. On the other hand, they must be concrete enough to compile them easily and efficiently onto the available platforms.

With the enormous increase in the complexity of software, the concept of separation of concerns have become more and more important. Naturally, the first class abstractions of the languages have become the essential way of expressing the separation of concerns in programs. They become important factors in evaluating languages because they define the direct support that a programming language offers for expressing concerns in programs.

In the following sections, to analyze the programming languages historically, we will characterize them with respect to their first-class abstractions, and we will historically categorize the de-

* The author is supported by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

MODULARITY Companion'15, March 16–19, 2015, Fort Collins, CO, USA
Copyright 2015 ACM 978-1-4503-3283-5/15/03...$15.00
http://dx.doi.org/10.1145/2735386.2735387

velopment of imperative languages as procedural, object-oriented (OO), and aspect-oriented (AO). Although this development helped to increase the liberation of programming languages from the von Neumann architecture, in software engineering, the semantics of problem domain is mainly defined in terms of emergent behavioral patterns, which cannot effectively be represented and modularized by these languages [6–10]. We outline a set of requirements to overcome these shortcomings, and explain the concept of **event-based modularization** as a solution. We explain that this concept helps to liberate programs from the von Neumann architecture further through flexible definition of modules as groups of events.

2. A Historical Perspective to Programming Languages

As shown in Figure 1, in procedural languages, a program is divided into a set of procedures/functions, which invoke each other with zero or more call arguments. The callee procedure/- function executes the call, and it may invoke on other procedures; this form of communication is known as the client/server communication.

Compiling procedural languages onto von Neumann architecture is relatively straightforward: the abstractions of an imperative program: procedures, statements, input-output parameters have a direct correspondence to von Neumann instructions and addressable memory locations. This enables efficient execution of programs and relatively effective implementation of compilers. Procedural languages, therefore, have a strong machine-oriented flavor.

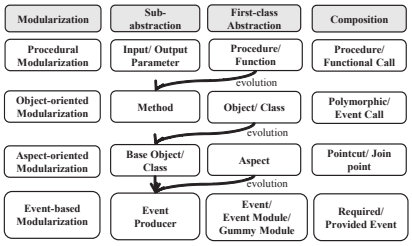

Figure 1: The evolution of four modularization mechanisms

The software systems of that time were mainly applied to solving well-structured and relatively simple problems. The platform technology did not have the processing power to address more complex problems and to support more complex programming abstractions. When the processors became more powerful, there was an opportunity to address more complicated problems in software. Accordingly, software systems have become increasingly more complex and expensive. Reducing complexity and enhancing flexibility and evolvability of software systems have been considered as important requirements.

To liberate the programmers from machine-oriented view of programming, object-oriented (OO) languages have been introduced. These languages provided objects/classes as a means to structure software systems. The roots of OO languages go back to Simula, which is a language for simulating complex systems. Objects were considered a better match for the abstractions of the real world. An object groups a set of attributes and procedures (methods) together. The methods can be explicitly invoked, or if the language offer an event-delegate mechanism, they can be invoked implicitly via event announcement.

As a structural reuse mechanism, objects can be typed, and reused polymorphically using the class and class inheritance constructs. In the consequent years, many OO languages have been introduced and this has resulted in the vast adoption OO languages by industry. Polymorphic calls and hierarchical organization of objects made OO programs look quite different from von Neumann architectures. On the other hand, the object structure is somehow similar to a memory location in a von Neumann architecture which is addressable by dedicated operations.

OO languages are particularly useful in representing hierarchically structured entities of the real world. However, OO programs in some cases creates too rigid program structures. Therefore design-patterns have been introduced to bring some flexibility to the programs. For example, the Abstract Factory pattern was proposed to increase the flexibility in instantiating objects with different types; the Bridge pattern was introduced to support objects with flexible structures. In certain cases, the programmers had to codify many patterns in the same program piece, which led to increased complexity of software. As such, implementation of patterns looked more like workarounds than meaningful architectural abstractions. Moreover, OO languages fail short in representing non-hierarchical structures. As a consequence, implementation of certain concerns, such as tracing of executions of non-hierarchically organized objects, becomes problematic in OO programs; tracing code tangles with and scatters across the implementation of other concerns in multiple objects. This increases the complexity and reduced the flexibility and evolvability of programs.

Aspect-oriented (AO) programming languages have been introduced to represent non-hierarchical structures, and as such they offer more powerful abstractions to implement concerns like tracing. To this aim, AO programing languages extends the OO model through implicit call mechanisms; objects do not need to refer each other through explicit names but through quantification predicates. This provides a more reusable and flexible coupling among the caller and callee procedures, especially when the procedures are not related to each other through a (class) hierarchy. By the help of implicit invocation mechanisms, AO programming languages liberate the programmer from von Neumann architectures even better than OO languages.

Along with the struggle of liberating programmers from von Neumann architecture, it is clear that all the relevant concerns in the problem domain must be effectively represented by the first class abstractions of the languages. With the introduction of AO languages, the liberation effort has been successful along the structural dimension, from procedures to objects, and from hierarchical organization of objects to flexible organizations of objects (aspects). However, in software engineering, the semantics of problem domain is mainly defined in terms of emergent behavioral patterns, which cannot effectively be represented by current languages [6–10].

3. Study on Emergent Behavior

Modelling behavior and patterns associated with behavior have been studied in various disciplines such as biology and behavioral sciences, system theory and traditional engineering domains. In computer science, study on behavior is mainly carried out in the definition of the semantics of programs. Since software systems can be practically applied to wide area of domains, we will now consider the concept of behavior and behavioral patterns from a wider perspective. This would help us to determine the desired characteristics of the first-class abstractions of new generation of programming languages.

Emergent behavior is generally defined as the appearance of complex behavior out of multiplicity of relatively simple interactions among its constituents [4]; it is a macroscopic effect that is caused by the microscopic interactions of its simpler constituents.

Various classifications and forms of emergent behavior have been defined in the literature. In this paper, we consider the classification proposed in [4]. At its simplest form, emergent behavior refers to the intentional design of a system to expose certain behavior. In this type, fixed roles are assigned to the constituents forming

the system, and there is no feedback from emergent behavior to the constituents to adapt themselves. More complex form of emergent behavior appears from the interactions of constituents, and in turn influences the interactions and behavior of the constituents through a feedback process. An example of this case is traffic congestion, which is a behavior formed from the interactions of multiple drivers, and in turn it influences the way the drivers continue driving. There can be multiple levels of emergent behavior in complex adaptive systems, where there are multiple levels of feedback process. An example, is the emergence of air pollution in an area where traffic congestion has appeared.

In biology and behavioral sciences, for example, seedling, behavior of colonies of ants, piles of termites, swarms of bees, flocks of birds, herds of mammals, shoals/schools of fish, packs of wolves have been studied. In system sciences and engineering, different kinds of emergent behavior have been considered in modelling, simulating and controlling of road and air traffic systems, transport systems, nature's patterns, etc.

In software engineering, we look at emergent behavior from two perspectives. First, the behavior that software exposes is regarded as emergent behavior [4]. Here, specific constituents (e.g. statements) are put together to exhibit the behavior; it is not easy or even possible to reduce the behavior of software to the behavior of its individual constituents. Second, nowadays, there are various kinds of software systems that deal with detecting the emergence of certain behavior in environment, representing it in the software and providing means to manipulate the behavior. Self-adaptive software systems, runtime verification techniques, various monitoring and simulation systems such as traffic monitoring/ simulation systems, and cyber physical systems such as air traffic control systems are typical examples.

4. Designing Effective Programming Languages

The initial step to deal with in software engineering is to develop suitable algorithms to detect the appearance and disappearance of the emergent behavior, as well as algorithms to manipulate the emergent behavior in software. In addition to algorithmic focus on emergent behavior, it is necessary to provide suitable programming languages and module abstractions to implement the algorithms and to modularize the implementations, respectively. We define the following set of basic requirements for the first class abstractions of programming languages:

Events and states should be the basic abstractions: The semantics of emergent behavior may be expressed in various formalisms; typical examples are regular expressions, state machines, first-order logic based expressions, temporal logics, etc. Regardless of the adopted formalism, the notions of **events** and **states** can be seen as two fundamental concepts in representing emergent behavior. An event represents a state change of interest, and can be regarded as a means to abstract necessary information about the state change. A state represents the value of all the stored information within a given context at a given instant in time. We see emergent behavior as a sequence of state changes, which is specified in certain formalism(s).

The notions of events and states are also fundamental concepts in language models. For example, invocation and/or execution of procedures in procedural languages, evaluation of logical predicates in logical languages, invocation and/or execution of methods on objects, object construction and destruction in OO languages, and the activation of joint points in AO languages, can all be regarded as events. Current languages support various means to define states of interest; for example, through local and/or global variables.

The module concept is useful in abstracting complexity and in building high-level abstractions: Where events and states are the core abstractions of emergent behavior, they can be too low level for representing the concerns of interest in software. Modules can be seen as means to increase the granularity of representations through representing individual concerns as first-class entities in software.

Module interfaces should be (also) defined as a set of events but not necessarily as a set of methods: It is generally accepted that a module has well-defined provided and required interfaces, and encapsulates its implementation. In an event-based view of emergent behavior, modules can be seen as a certain composition of events, which encapsulate relevant state information. The required interface of modules specifies the set of events to which the modules reacts; the implementation part of the modules defines the semantics for processing these events; the modules may publish their internal states as events, which form their provided interface. This way of modularization can help in expressing the relevant concerns of emergent behavior as the first-class abstractions of the language.

Events must not be fixed but user-definable: The set of necessary events to represent emergent behavior of software is open-ended and is influenced by application domain. For example, in one application we may require to deal with events of operating system processes, external events collected from hardware components and users, and events published by software modules. Since events are in principle open ended, it must be possible to define interfaces of the modules flexibly through associative composition of events.

Different module implementation techniques/formalisms must be supported: It is desirable to keep the interfaces of modules separated from the implementations of the modules, so that the implementations can be flexibly adapted. Besides, the language in which the implementation part of modules must be expressed cannot be fixed in general. Various domain-specific languages and formalisms, such as state machines and temporal logics formula, as well as general-purpose languages can be adopted for this matter.

Various dedicated languages have been defined which did not follow the imperative language paradigm, such as functional languages, logic-based languages, data-flow languages, etc. In [2], for example, functional programming languages were proposed to liberate the programmers from the von Neumann architecture. None of these proposals fulfill the requirements outlined in this paper, as such they lack language abstractions to modularize behavioral patterns in some application domains. Furthermore, within the context imperative languages, we have extensively studied the shortcomings of various general purpose and domain-specific languages in fulfilling the requirements outlined in this paper [6–10].

5. Event-based Modularization

In the von Neumann architecture, the basic elements are defined as statements (functions), registers and memory locations, instruction fetch and decoding controlled by the program counter. Although events can be supported by the von Neumann hardware in terms of interrupts, if they need to be expressed in software, they have to be programmed. The abstractions are based on statements, registers and memory locations but not as a group of events. The behavior is expressed by steering the program counter.

Considering events and states as the basic abstractions, supporting arbitrary kinds of events and state abstractions, grouping sets of relevant events as modules, and using logical expressions (like predicates) as the notations of events and states, would make programs deviate from the basic mechanisms of von Neumann architecture considerably.

We have been developing the concept of **event-based modularization** [6–9], which offers first-class abstractions to overcome these shortcomings. As Figure 2 shows, at a high level of abstraction, we consider the environment as a set of **events**, which may

be published by the objects and aspects that exist in an application, and/or by external entities such as OS and hardware drivers. Events are typed entities; an event type defines a set of attributes for the events.

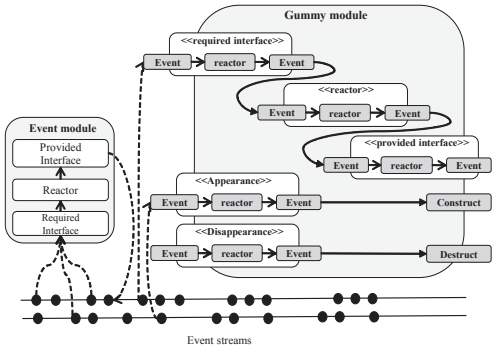

Figure 2: Event and gummy modules

We introduced **event modules** [6, 7] and **gummy modules** [8, 10] as means to modularize a group of related events and the reactions to them. These modules have required and provided interfaces, which specify the set of events they select from the environment and publish to the environment, respectively. The implementation part of the modules, which is termed as **reactor**, provide the functionality to process and publish the events.

A gummy module with primitive interfaces and reactor is equivalent to an event module. A composite gummy module consists of a set of (primitive) gummy modules as its required interface, provided interface, and/or reactor. Composite gummy modules are means for encapsulating a group of correlated gummy modules/ event modules. A gummy module may be instantiated explicitly like an object, or it may encapsulate its construction and destruction semantics through its appearance and disappearance code blocks. The latter is termed as **emergent gummy module** [8], and is suitable to modularly represent emergent behavior that has transient nature.

The events published by an event/gummy module can be selected further by other event/gummy modules. This facilitates the composition of event/gummy modules with each other, and modularly expressing composition constraints as event/gummy modules in their desired language [8, 10].

Event-based modularization does not fix the set of supported events and even; new kinds of event types, attributes and events can be defined according to applications demands. The languages in which the interfaces and the reactors can be programmed are not fixed; various domain-specific and general-purpose languages can be adopted for this matter. In event/gummy modules, the interfaces and implementations of modules are separated from each other.

We have shown that these features paves the way to flexibly modularize various kinds of concerns in software, and to achieve better adaptability in modules [6, 7].

6. Achieving Uniformity in Modularization

As the complexity of today's software systems increases, it is becoming inevitable to design them as systems of systems, where each system may offer functionalities specific to a certain domain. Besides, each system may provide additional functionalities, such as monitoring external environment, self-adaptation and runtime verification, to increase the reliability of its behavior in changing environments. These imply that new categories of concerns may appear in software systems, which must effectively be represented and modularized.

As a result, the evolution of modularization mechanisms will continue, and we will have software systems that are modularized via various modularization mechanisms; for example, real-world entities, crosscutting concerns and emergent behavior are represented via objects, aspects, and event/gummy modules, respectively. Diversity in module abstractions reduces the composeability of software if suitable mechanisms are not available to compose diverse kinds of modules with each other. Therefore, it is of interest to achieve uniformity in representation of concerns in software.

We claim that the notion of emergent behavior plays an important role in achieving such a uniformity. Software engineers design and implement software to expose certain behavior, by putting specific constituents (e.g. statements) together to exhibit the behavior. Since it is not easy or even possible to reduce the behavior of software to the behavior of its individual constituents, software behavior can be regarded as emergent behavior [4]. Based on this definition, design and implementation of software imply design and implementation of known and desirable emergent behavior; modularization of software behavior imply modularizing emergent behavior.

Event-based modularization is where modularity and event-driven programming meet to facilitate representing various kinds of concerns. Event-based modularization extends the traditional view on events, which mainly considers events as means for asynchronous communication. It considers events as the core abstractions of computations, which must be taken into account as the core abstractions for defining modules. Emergent behavior is an example kind of concern, which should be modularized. As we explained in Section 4, regardless of the adopted formalism, the notions of events and states are fundamental in representing the semantics of emergent behavior, let it be the emergent behavior of software or the emergent behavior in an external environment. Event/gummy modules are flexible module abstractions to represent emergent behavior.

References

[1] M. Aksit. The 7 C's for Creating Living Software: A Research Perspective for Quality-Oriented Software Engineering. *Turkish Journal of Electrical Engineering & Computer Sciences*, 12:61–95, 2004.

[2] J. Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 1978.

[3] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.

[4] J. Fromm. Types and Forms of Emergence. URL `arXiv:nlin/0506028`.

[5] K. C. Louden. *Programming Languages: Principles and Practice*. PWS-Kent Publishing Company, 1993.

[6] S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. PhD thesis, University of Twente, 2011.

[7] S. Malakuti and M. Akşit. Event Modules: Modularizing Domain-specific Crosscutting RV Concerns. In *Transactions on Aspect-Oriented Software Development XI*, volume 8400, pages 27–69. LNCS, 2014.

[8] S. Malakuti and M. Aksit. Emergent Gummy Modules: Modular Representation of Emergent Behavior. In *GPCE '14*. ACM, 2014.

[9] S. Malakuti and M. Aksit. Event-Based Modularization of Reactive Systems. In *Concurrent Objects and Beyond*, volume 8665 of *LNCS*, pages 367–407. 2014.

[10] S. Malakuti and M. Aksit. Event-based Modularization: How Emergent Behavioral Patterns Must Be Modularized? In *FOAL*. ACM, 2014.