Science of Computer Programming 98 (2015) 120–139

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A feature model of actor, agent, functional, object, and procedural programming languages

Howell Jordan [a,*], Goetz Botterweck [a], John Noll [a], Andrew Butterfield [b], Rem Collier [c]

[a] Lero, University of Limerick, Ireland
[b] Trinity College Dublin, Dublin 2, Ireland
[c] University College Dublin, Belfield, Dublin 4, Ireland

HIGHLIGHTS

- A survey of existing programming language comparisons and comparison techniques.
- Definitions of actor, agent, functional, object, and procedural programming concepts.
- A feature model of general-purpose programming languages.
- Mappings from five languages (C, Erlang, Haskell, Jason, and Java) to this model.

ARTICLE INFO

Article history:
Received 9 March 2013
Received in revised form 31 January 2014
Accepted 5 February 2014
Available online 20 February 2014

Keywords:
Programming languages
Programming language constructs
Agent-oriented programming
Functional programming
Object-oriented programming

ABSTRACT

The number of programming languages is large and steadily increasing. However, little structured information and empirical evidence is available to help software engineers assess the suitability of a language for a particular development project or software architecture.

We argue that these shortages are partly due to a lack of high-level, objective programming language feature assessment criteria: existing advice to practitioners is often based on ill-defined notions of 'paradigms' [3, p. xiii] and 'orientation', while researchers lack a shared common basis for generalisation and synthesis of empirical results.

This paper presents a feature model constructed from the programmer's perspective, which can be used to precisely compare general-purpose programming languages in the actor-oriented, agent-oriented, functional, object-oriented, and procedural categories. The feature model is derived from the existing literature on general concepts of programming, and validated with concrete mappings of well-known languages in each of these categories. The model is intended to act as a tool for both practitioners and researchers, to facilitate both further high-level comparative studies of programming languages, and detailed investigations of feature usage and efficacy in specific development contexts.

1. Introduction

Programming languages are traditionally viewed as belonging to particular paradigms, however the notions of programming paradigms [3, p. xiii] and orientation [4] are imprecise. Unlike scientific paradigms [5, p. 148], programming paradigms

* Corresponding author.
E-mail addresses: howell.jordan@lero.ie (H. Jordan), goetz.botterweck@lero.ie (G. Botterweck), john.noll@lero.ie (J. Noll), andrew.butterfield@scss.tcd.ie (A. Butterfield), rem.collier@ucd.ie (R. Collier).

http://dx.doi.org/10.1016/j.scico.2014.02.009
0167-6423/

are not necessarily incompatible, as demonstrated by the success of dual- and multi-paradigm languages such as Mozart/Oz (http://www.mozart-oz.org), Jason (http://jason.sf.net), and Scala (http://www.scala-lang.org). This paper attempts to identify, define, and organise the central concepts underlying the actor, agent, functional, object, and procedural programming styles, as they are realised in practical programming languages.

This paper has three central aims. Firstly, by mapping existing programming languages to a common feature model, it is hoped that ideas for new language features and new combinations of features will be generated. Secondly, it is hoped that the resulting feature model will serve as a basis for comparison and generalisation in empirical studies of multiple programming languages. Finally, the number of programming languages is large [1] and steadily increasing [2]. This model and its associated empirical evidence should eventually become a useful tool to help software engineers in assessing the suitability of a language for a given development project or software architecture.

With these second and third aims in mind, the languages in this paper were selected as popular examples of their respective 'paradigms'. Programming language popularity is hard to measure, however we have used the listing at http://www.tiobe.com/index.php/content/paperinfo/tpci (accessed February 2013) as a guide. C [6] is probably the most popular procedural programming language. Erlang (http://www.erlang.org) [7] is a functional language with a rich industrial heritage [8], based on the actor model of concurrency [9,10]. Haskell (http://www.haskell.org) is a purely-functional language. Jason [11] is an agent-oriented language [12] which implements and extends AgentSpeak(L) [13]. Java (http://www.oracle.com/technetwork/java) is probably the most popular object-oriented programming language.

The reference versions of each programming language considered here are Erlang R13B03, Haskell 2010 (as implemented by the Glasgow Haskell Compiler version 7.0.4), Jason 1.3.4, and Oracle Java 1.7.0.40. Unfortunately, at the time of writing, many of the most popular C compilers do not fully implement the most recent C standards. In particular, the GCC (http://gcc.gnu.org) and Microsoft Visual Studio (http://www.microsoft.com/visualstudio) compilers do not fully implement either C99 [14] or C11 [15]. Consequently, the reference version of C adopted for this paper is C90 [16] (also sometimes known as ANSI C or C89), which is supported by the above compilers and is the version discussed in the well-known reference by Kernighan and Ritchie [6]. Due to C's heritage as a systems programming language, several important features not included in the core language are provided instead by platform libraries which are defined in the separate Portable Operating System Interface (POSIX) standards [17]. As implementations of these libraries are provided 'out-of-the-box' on many platforms, we have considered them as part of the C language where appropriate.

The remainder of this paper is structured as follows. Section 2 introduces feature modelling. Section 3 presents some examples of existing feature-based surveys, and provides an overview of comparisons and concepts of programming languages. In Section 4, a feature model of actor, agent, functional, object, and procedural programming languages is developed from the literature and validated against the languages listed above. Section 5 concludes, discusses the limitations of the feature model, and suggests several directions for further work.

2. Feature modelling

Feature modelling supports the informal comparison of existing and future systems, by characterising systems and their features as instances of domain concepts [18, Ch. 4]. Apel and Kästner [19] identify ten different definitions of the term *feature*, reflecting the fact that feature modelling can be applied at many stages of the software lifecycle, and at levels of granularity ranging from domain analysis [20] to compile-time configuration of operating systems [21].

Feature modelling is commonly used to manage variability in the context of software product lines [22,23]. However, the focus of this paper is on the feature-oriented domain analysis of high-level application programming languages, with the objective of defining "the features and capabilities of a class of related software systems" [20]. Feature modelling is a creative activity [18, p. 85] which is often also iterative and community-driven.

Feature-based comparisons incorporate many ideas from earlier classifications and taxonomies, with an added emphasis on optimising models so as to maximise composability, reduce dependencies between features, and thus minimise feature interactions [24]. Feature-based and framework-based comparison studies share several key characteristics: the central objective of both study types is to integrate selected work within a pre-defined boundary, to produce a single cohesive model [25]. Unlike reviews, which aim to be comprehensive, framework- and feature-based comparisons typically focus on higher-level concepts and the relationships between them.

An abstract model of a product family, such as a feature model, can be assessed either by studying one product instance in its intended context, or by analysing a subset of product instances with respect to the model [26, p. 206]. In this paper, the latter approach is adopted; each product instance is a well-known programming language. When selecting product instances for model assessment, there are two possible strategies. Typically, products describing the extremes are selected; alternatively, product popularity may be used as a selection criterion [26, p. 206]. The products selected for inclusion in this study were chosen because they are both popular and widely spaced within the programming languages landscape.

The terms *concept*, *characteristic*, and *feature* are used in this paper as follows. A *concept* is loosely defined as any idea or principle, often (but not necessarily) based on or utilised in theory. A *feature* is a realisation of a concept within the context of a family of related systems (in this case, programming languages), and a *feature instance* is a realisation of a feature in a specific system (in this case, a particular language). A *characteristic* is an observable property of a system or feature instance.