A Feature Model of Actor, Agent, and Object Programming Languages

Howell Jordan
Lero @ University College Dublin
howell.jordan@lero.ie

Goetz Botterweck
Lero @ University of Limerick
goetz.botterweck@lero.ie

Marc-Philippe Huget
University of Savoie
marc-philippe.huget@univ-savoie.fr

Rem Collier
University College Dublin
rem.collier@ucd.ie

Abstract

This paper presents first steps towards a feature model, which can be used to compare actor-oriented, agent-oriented, and object-oriented programming languages. The feature model is derived from the existing literature on general concepts of programming, and validated against Erlang, Jason, and Java. The model acts as a tool to assist practitioners in selecting the most appropriate programming language for a given task, and is expected to form the basis of further high-level comparative studies in this area.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

General Terms Languages

Keywords Actor model, Agent-oriented programming, Object-oriented programming, Feature modelling

1. Introduction

Programming languages are traditionally viewed as belonging to particular paradigms, however the notion of a programming paradigm is imprecise [43, p.xiii]. In particular, with the advent of multi- and dual-paradigm languages, such as Mozart/Oz[1], Jason[2], and Scala[3], it is clear that many programming paradigms do not meet the classic definition of a scientific paradigm [26, p.148]: they are not "incommensurable". This paper attempts to bridge the gap between the three mostly-separate bodies of literature, by focusing on the concepts underlying the actor, agent, and object programming styles - and their realizations as features in practical programming languages.

This paper has two 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, the resulting feature model can help software engineers to select the most appropriate language for a given problem.

With this second aim in mind, the languages in this paper were selected as popular examples of their respective paradigms. Erlang[4] [5] is a functional language with a rich industrial heritage [4], based on the actor model of concurrency [22, 2]. Jason [7] is an agent-oriented language [37] which implements and extends AgentSpeak(L) [34]. Java[5] is probably the world's most popular[6] object-oriented programming language.

The remainder of this paper is structured as follows. Section 2 gives an overview of related work in feature modelling and in programming language concepts. In Section 3, a feature model of actor-, agent-, and object-oriented programming languages is developed from the literature and validated against the languages listed above. A high-level overview of this feature model is shown in Figure 1. Section 4 concludes, discusses the limitations of the feature model, and suggests several directions for further work.

2. Related Work

We are not aware of any existing feature models or feature-based surveys of programming languages. Consequently this section first discusses the feature modelling technique, and some examples of its use in related areas. Then a brief overview of the literature on the underlying concepts of programming languages is presented.

2.1 Feature Modelling

Feature modelling supports the informal comparison of existing systems, by characterising systems and their features as instances of domain concepts[7] [10, ch.4]. Feature modelling is a creative activity [10, p.85] which is often also iterative and community-

[1] http://www.mozart-oz.org

[2] http://jason.sf.net

[3] http://www.scala-lang.org

[4] http://www.erlang.org

[5] http://www.oracle.com/technetwork/java

[6] Programming language popularity is hard to measure, however as of August 2011, Java was the top object oriented programming language listed at http://langpop.com and http://www.tiobe.com/index.php/content/paperinfo/tpci.

[7] The terms 'concept', 'characteristic', and 'feature' appear frequently in this paper; we use them as follows. A concept is loosely defined as any idea or principle, often (but not necessarily) based on theory. A feature is a realization of a concept within the context of a family of related systems (in this case, programming languages), and a feature instance is a realization

SPLASH'11 Workshops, October 23–24, 2011, Portland, Oregon, USA.

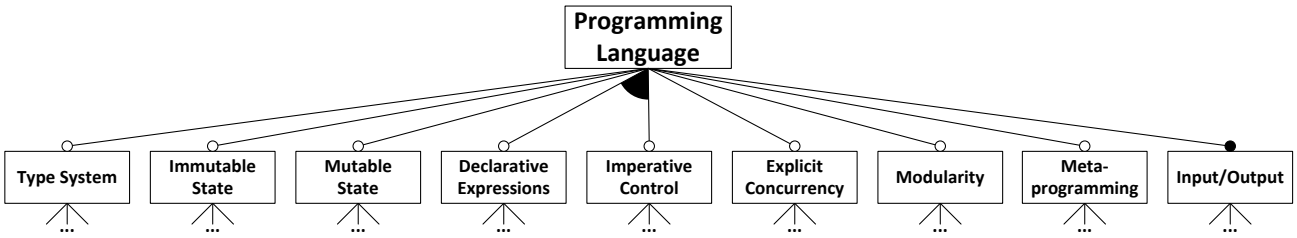

Figure 1: Overview of a feature model of actor, agent, and object programming languages.

driven. For example, Czarnecki and Helsen [12] provide a survey of model transformation languages, which extends and improves a feature model proposed in an earlier paper [11].

Feature modelling incorporates many ideas from earlier classifications and taxonomies, with an added emphasis on optimising the model so as to maximize composability, reducing dependencies between features and thus minimizing feature interactions [25]. Martin et al. [28] survey distributed computing systems using a taxonomic approach that is very similar to feature modelling. The emphasis of their survey is "breadth rather than depth", and the focus is on fundamental system features and their possible combinations. A taxonomic approach is also employed by Meier and Cahill [29] to survey the features of distributed event systems.

A feature model shares many of the key characteristics of a framework article. The central objective of a framework study is to integrate selected work within a pre-defined boundary, to produce a single cohesive model [35]. Unlike a review, which aims to be comprehensive, the focus of a framework is typically on higher-level concepts and the relationships between them.

2.2 Programming Language Concepts

Detailed discussions of the general concepts of programming languages are found in several well-known texts [1, 17, 43, 36, 41]. A classic overview of some fundamentals is provided by Strachey [38], while a more recent perspective may be found in Van Roy [42]. In specific subject areas, Tratt [40] surveys type system concepts, and Gabbay et al. [20] present the theoretical foundations of logic programming.

Dennis et al. [14, 15] adopt a theoretical approach to analyse the concepts underlying agent-oriented programming. Their framework, based on operational semantics, successfully models the core functionality of the well-known 3APL [13], AgentSpeak [34], and MetateM [18] languages, and leads to the identification and abstract specification of some 'missing' modularity features.

Hudak [23] introduces the key features of functional programming languages, and the lambda calculus. While Erlang is not a purely functional language, it implements several of these concepts. A more general theoretical treatment of declarative languages by Hanus [21] describes attempts to unify functional programming with logic languages (which share some important similarities with Jason) and the constraint programming paradigm. Armstrong [3] employs an empirical method to discover the fundamental concepts (called 'quarks') of object-oriented programming, which are then surveyed.

3. Feature Model

Due to the different terminology used in the actor, agent, and object literature, the domain concepts on which this feature model is based are drawn where possible from the wider literature on computer programming. It must be emphasised that the feature model presented here is neither final nor definitive; it can and should be modified and extended to accommodate new languages and developments.

A programming language may be modelled as one or more feature sets S_i, where S_i consists of feature instances $I_0 \ldots I_n$. Each feature instance I_i realizes, with a concrete syntax and semantics, one of the abstract features described in this section.[8] A complex programming language which consists of several interrelated sub-languages, may be better modelled as a set of feature sets $S_0 \ldots S_n$.

The feature model takes the form of a tree, with nine first-level nodes which are presented here as separate sub-trees. Features and feature categories are either mandatory (denoted by solid bubbles) or optional (denoted by hollow bubbles). The colours used in the feature diagrams that follow carry no semantics; their only purpose is to improve readability. Mappings from the three selected languages to the feature model are described in the accompanying tables.[9] A `monospace` font is used to provide examples of feature instances where space allows; the examples were tested against Erlang R13B03, Jason 1.3.4, and Java 1.6.0.26. *Italics* indicate language-specific terminology used in `http://www.erlang.org/doc`, Bordini et al. [7], and `http://download.oracle.com/javase/tutorial`.

3.1 Type System

Type systems serve two related purposes in programming languages: to classify values, and to determine their applicable operations. In the first view, a type is "a constraint which defines the set of valid values which conform to it" [40]. In the second, types are abstract "specifications of functionality" [41, p.723], which define "legal usage contexts for the values they describe" [41, p.620]. An attempt to perform an illegal operation on a value is known as a type error, which may be detected either at compile time or runtime. In a typeless language, "it must be the case that every value can be used in every context" [41, p.622].

Checking Type information may be checked for errors at compile time ('static' typing), at runtime ('dynamic' typing), or both [41, p.623].

Inference The information required for compile-time type checks can either be supplied explicitly by the programmer, or inferred by the compiler or interpreter using type reconstruction techniques [41, ch.13].

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.

[8] A single language may offer multiple realizations of the same abstract feature, usually for reasons outside the scope of this paper - for example to support backwards compatibility or to provide different performance characteristics.

[9] Where a language offers multiple instances of the same high-level abstract feature, and these instances exhibit significant feature differences at a lower level, these alternatives are presented using additional columns.

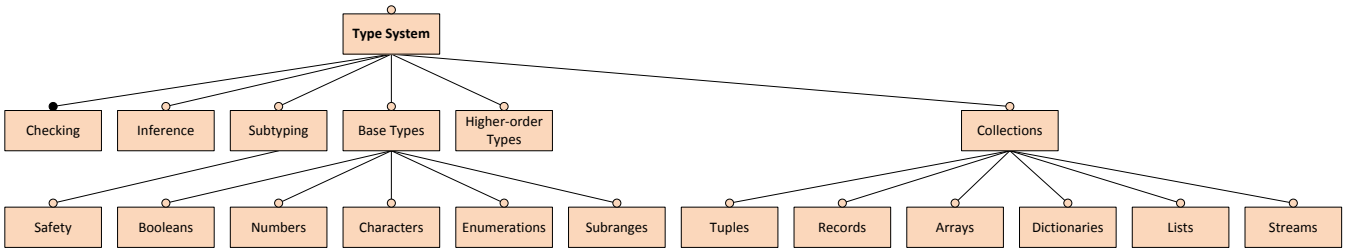

Figure 3.1: Type system features.

	Erlang	Jason	Java
Checking	Runtime [a]	Runtime	Compile time [b]
Inference	N/A	N/A	✗
Subtyping	✗	✗	`class A extends B {}`
Safety			Casting with runtime checks
Base Types			
Booleans	✗	✗	`boolean`
Numbers	*Integer*, *float*	*number*	`byte`, `short`, `int`, `long`, `float`, `double`
Characters	ASCII code integer	Single-character *string* `"s"`	`char`
Enumerations	✗	✗	`enum Compass {N,S,E,W}`
Subranges	✗	✗	✗
Higher-order Types	*Anonymous function*: `F=fun()->`	*Plans* are strings: `P="+b <-`	✗ [c]
Collections			
Tuples	`T={a,b}`	✗	✗
Records	`-record(r,a,b)` [d]	✗	`class R{final int a,b;}`
Arrays	✗	✗	`int[] a={4,2}` also *List* classes
Dictionaries	See Table 3.3 [e]	See Table 3.3 [f]	`class D{int a,b;}` also *Map* classes
Lists	`L=[a,b]`	`L=[a,b]`	✗ but defined in *Collection* interface
Streams	*Ports* for external I/O only	✗	*InputStream*, *OutputStream*

Table 3.1: Type systems in Erlang, Jason, and Java.

[a] Erlang code may also be checked at compile time for type errors using Dialyzer (see `http://www.erlang.org/doc/man/dialyzer.html`), a tool based on type inference techniques.

[b] Some features of Java, such as *reflection*, require type checks to be delayed until runtime [41, p.624].

[c] The Java *reflection* API provides a `Method` type for handling subroutines, however its use has many disadvantages and is officially discouraged.

[d] Erlang records are compiled as tuples; consequently, field names are replaced by integer indices at runtime.

[e] Only a single *dictionary* is available to each Erlang process.

[f] Dictionaries can be built using *structures* in Jason, albeit with little support from the type system.

Subtyping Subtyping allows two types to be compatible, without being the same [41, ch.12].

Safety Under an unsafe type system, the programmer may force (or 'cast') values of one type to be considered as conformant with an incompatible type. Type-safe languages either disallow casting, or perform runtime checks to ensure that any casts do not subvert the type system [40].

Base Types Base types represent atomic (indivisible) values, and serve as collection members.

Booleans represent true or false.

Numbers of specified precision, signed and unsigned [36, p.294].

Characters can be encoded in ASCII, multilingual Unicode, or another encoding [36, p.295].

Enumerations are ordered sets of named elements [36, p.297].

Subranges "compose a contiguous subset of the values of some discrete base type" [36, p.298]. The compiler may generate code to dynamically check that subrange values lie within their specified ranges; alternatively, this checking may be performed by the interpreter.

Higher-order Types A language may define higher-order types to represent functions or subroutines, allowing them to be "passed as parameters, returned by functions, or stored in variables" [36, p.290]. If higher-order types are true first-class language constructs, new values for those types can be computed at runtime [36, p.508] (see Section 3.8).

Collections Collections are composites formed from one or more base types. In keeping with the above definition of type as usage context, they can be divided according to how individual elements are accessed, and whether in-place modification of elements is permitted [43, p.438]. The mutable array and dictionary collection types are only applicable to languages with mutable state (see Section 3.3).

Tuples are immutable, indexed by integers.

Records are also immutable, indexed by any literal.

Arrays are mutable, and integer-indexed.

Dictionaries are also mutable, indexed by any literal.

Lists are unindexed, of finite length.

Streams are unindexed and unbounded; commonly used for input/output (see Section 3.9) and concurrency (see Section 3.6.

Table 3.1 gives an overview of type system features in the selected languages.

3.2 Immutable State

A crucial feature of many programming languages is "the possibility of associating values with symbols and later retrieving them" [1, p.8]. This feature category is concerned with symbol-value associations which, once made, cannot be changed.

Constants A constant is defined here as a binding, between a symbol or name and a value, which lasts for the lifetime of the enclosing element.

Single Assignment A single assignment variable - sometimes known as a 'declarative variable' - is initially an unassigned symbol, but once bound "stays bound throughout the computation and is indistinguishable from its value" [43, p.42].

Table 3.2 outlines the immutable state features of Erlang, Jason, and Java.

3.3 Mutable State

Van Roy and Haridi [43, p.408] define the named state of a computational entity as "a sequence of values in time that contains the intermediate results of a desired computation". While not all programming languages provide explicit state representation, any entity that is aware of its past must store that knowledge either internally, or externally in the environment [43, p.410].

Declaration A declaration introduces a name and indicates its scope [36, ch.3]. The declaration may also include type information (see Section 3.1).

Assignment Assignment associates a state cell with a value. Some languages allow assignment between two state cells, in which case assignment may be either by value (the value of the second state cell is copied to the first) or by reference (the first cell is modified to refer to the second) [36, p.225].

Valuedness A state cell is multivalued if more than one value may be associated with a single name and index. Otherwise it is single valued.

Retrieval The mechanism by which the value of a state cell is retrieved, given a name.

Modification A language may provide special constructs to reassign the value of a state cell.

Deletion The mechanism by which a state cell, and its contents, are erased.[10]

Table 3.3 describes the mutable state features of the selected languages.

[10] Recovery of the resources underlying the deleted state cell, for example by garbage collection, is considered an operating system issue and therefore out of scope for this paper.

3.4 Declarative Expressions

Finkel and Kamin [17] define declarative programming as a separation of logic and control. The programmer creates the logic component, consisting of declarative expressions, which "specifies what the result of the algorithm is to be" [17, p.238]. The control component is partly or wholly provided by the compiler or interpreter.

Functions A function is a procedure which computes a mathematical function: two evaluations with the same arguments will always produce the same result [1, p.230]. This property is known as 'referential transparency' [23, p.362]. A function is distinct from a subroutine (see Section 3.5), in that side-effects are not permitted.

Evaluation The result of a function can sometimes depend on how its arguments are evaluated. A function may be evaluated in applicative order (evaluate the arguments, then apply the function) or normal order (fully expand the function until only primitive operators remain, then reduce) [1, p.16].[11] Under normal order evaluation, a function may return a value even if evaluation of some of its arguments would produce errors or not terminate [1, p.400]. However, implemented naively, normal order evaluation is inefficient and causes unnecessary repeated computations. A third option, lazy evaluation, avoids these recomputations by ensuring that all arguments are evaluated no more than once [23, p.383]. The result of a computation under normal order or lazy evaluation may also depend on the order of function arguments [23, p.390].

Conditionals Conditional expressions allow discontinuous functions to be defined, and are central to an 'equational reasoning' programming style [23, p.388]. The predicate-consequent pairs of a conditional may be evaluated in a given order, or the language may insist that the predicates in a conditional are disjoint.

Tail Recursion An expression is known as a 'tail call' if no computational work is done between the termination of the expression and the termination of its enclosing function [41, p.1044]. A tail recursive language guarantees that recursive tail calls will consume no additional memory resources, thus allowing iteration to be efficiently expressed. Tail recursion is supported in some languages with additional syntactic sugar [1, p.35].

Inference Rules Inference rules allow new knowledge to be derived from existing facts[12]. Inference rules are typically (though not necessarily) expressed as Horn clauses, consisting of an antecedent (potentially containing many terms) and a single-term consequent. Both antecedent and consequent terms may contain variables, thus allowing general relations to be expressed.

Resolution Resolution is the runtime process by which new knowledge is derived from inference rules: antecedent terms are matched ('unified') with facts and the consequents of

[11] Analogously, a function or individual argument may be described as strict or nonstrict. The value of a strict function is only defined if the values of all its arguments are defined [36, p.523]; if a particular argument is evaluated before the function body is entered, the function is strict in that argument [1, p.400]. While evaluation order is a property of the programming language, some languages place strictness under the programmer's control, so the former terminology is preferred here.

[12] A 'fact' (in Prolog terminology) is a Horn clause with no antecedent terms and no unbound variables. However, in many newer languages, facts are instances of mutable state (see Section 3.3). Efficient resolution in the presence of mutable state requires 'truth maintenance' mechanisms to ensure that previously-derived knowledge is only updated when necessary.

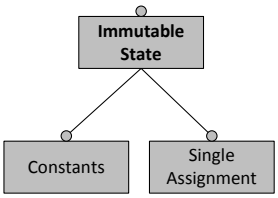

Figure 3.2: Immutable state features.

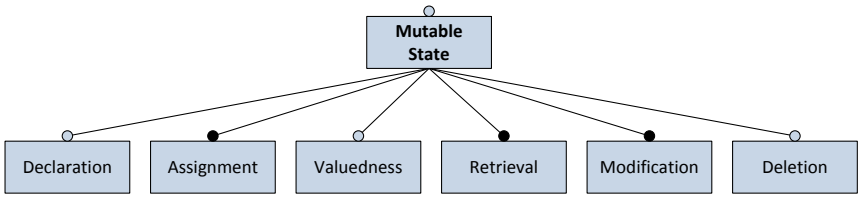

Figure 3.3: Mutable state features.

	Erlang	Jason	Java
Constants	`-define(C,2)`	Only in *plans*. `C=2`	`final int c=2` [a]
Single Assignment	Only in *functions*. `A=2`	Only in *plans*. `A=2`	Only in *constructors*. `final int a; a=2`

Table 3.2: Immutable state in Erlang, Jason, and Java.

[a] The Java `final` keyword does not protect members of collections and mutable compound types from modification.

	Erlang	Jason	Java	Java *Map* Library Classes
Declaration	✗	✗	`int a`	✗ [a]
Assignment	By value. `put(a,2)`	By value. `+a(2)`	By reference or value. [b] `a=2`	By reference or value. `m.put("a",2)`
Valuedness	Single	Multi. `+a(2); +a(3)`	Single	Single
Retrieval	`get(a)`	`?a(X)` [c]	`x=a`	`x=m.get("a")`
Modification	`put(a,3)`	`-a(2); +a(3)` or `-+a(3)` [d]	`a=3`	`m.put("a",3)`
Deletion	`erase(a)`	`-a(3)` or `-a(_)`	✗	`m.remove("a")`

Table 3.3: Mutable state in Erlang, Jason, and Java.

[a] State cells within a Java *Map* are not declared individually, however the map itself must first be initialised: `Map<String,Integer> m = new HashMap<String,Integer>()`.

[b] Java assignment is by value for primitive types (`int`, `double`, etc.) and by reference for instances of `Object`.

[c] The Jason interpreter attempts to match *test goals* against the agent's *beliefs* in reverse chronological order. The `include` preprocessor directive (see Section 3.7) unfortunately interacts with this feature [27]. If no match is found, a test goal addition event is generated (see Section 3.5).

[d] The Jason `-+` operator first removes all beliefs that match the given functor, then the new belief is added.

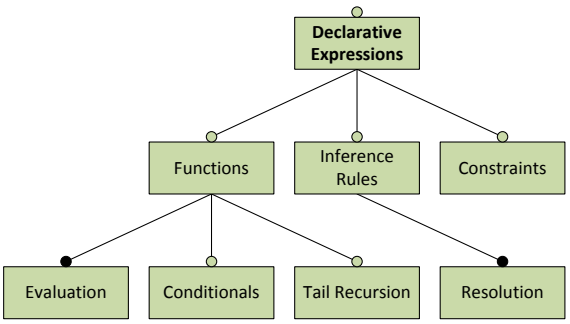

Figure 3.4: Declarative expression features.

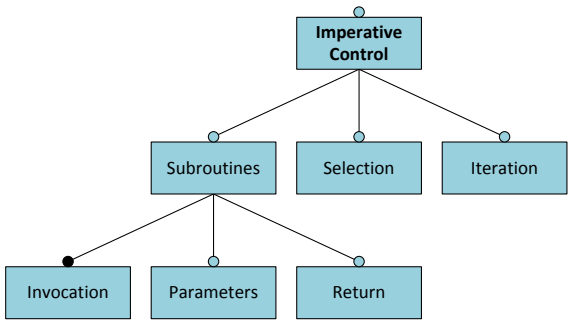

Figure 3.5: Imperative control features.

other inference rules, backtracking on failure, until no unmatched antecedent terms and unbound variables remain. Since multiple solutions may exist to any given knowledge base query, the order in which candidate facts and inference rules are selected determines the order in which solutions are found [17, p.235]. The resolution process is potentially recursive, and therefore must be executed in a defined order over antecedent terms, to prevent accidental nonterminating queries [36, p.554]. Resolution is usually carried out left to right, and either depth first or breadth first.

Constraints A constraint is a mathematical or logical relation between two or more variables, or a restriction on the domain of a variable, that must be satisfied.

3.5 Imperative Control

Imperative control allows the programmer to explicitly specify the execution or evaluation order of statements or expressions in time. Programmer-specified statement sequences are central to imperative languages [36, p.220], and are also required to support structured input/output (see Section 3.9) in declarative languages [33].

	Erlang	Jason	Java
Functions	`add(A,B) -> A+B.`	✗	✗
Evaluation	Applicative order		
Conditionals	`max(A,B) when A>B -> A; max(A,B) -> B.` [a]		
Tail Recursion	Automatic when last expression is a function		
Inference Rules	✗	Horn clause *rules*. `positive(X) :- .number(X) & X>0.`	
Resolution		Top-down, left to right, depth first [b]	
Constraints	✗	✗	

Table 3.4: Declarative expressions in Erlang, Jason, and Java.

[a] In addition to the *guard sequences* illustrated here, Erlang also provides familiar *if* and *case* expressions.

[b] Jason rules are resolved with the agent's beliefs in reverse chronological order (see Section 3.3).

	Erlang	Jason	Java
Subroutines	✗	*Plan*	*Method*
Invocation		By *triggering event* and *context*. `+!te : ?c <-`	By name and matching parameter types
Parameters		Positional, by value. `+!b(A,B)`	Positional, by reference or value.[a] `void s(int a,int b)`
Return		✗	Explicit, with termination. `return 2`
Selection		`if(X>Y) {+max(X)} else {+max(Y)}`	`if(x>y) {max=x;} else {max=y;}` [b]
Iteration		`while`, `for`	`while`, `do while`, `for`

Table 3.5: Imperative control in Erlang, Jason, and Java.

[a] Java method parameters are passed by value for primitive types (`int`, `double`, etc.) and by reference for instances of `Object`.

[b] Java also offers a `switch` statement, which selects between code fragments based on the value of a single variable, and supports *fall through*.

Subroutines A subroutine encapsulates a sequence of imperative control constructs, so they may be treated as a single unit [36, p.219]. A subroutine is distinct from a function (see Section 3.4): a subroutine may change the state of the program (see Section 3.3) or its environment (see Section 3.9) via side-effects.

Invocation The mechanism by which a subroutine is selected for invocation.

Parameters A subroutine may accept input data by declaring formal parameters, which are associated with arguments during invocation. Formal parameters and arguments may be associated by name, or by position [36, p.405]. Parameters may be passed by value (the argument is copied to the corresponding formal parameter) or by reference (the formal parameter is a new name for the corresponding argument) [36, p.395]; reference parameters may be used for output if the argument is mutable.

Return Return values allow a subroutine to send a result to its invocation context. The return mechanism may be implicit (the subroutine result is simply the value of its body), or the language may offer a formal 'result' parameter or an explicit return statement. Use of the return statement, or assignment of the result parameter, may also immediately terminate the subroutine [36, p.408].

Selection Allow choices between two or more code fragments, depending on runtime conditions [36, p.219].

Iteration Allow a code fragment to be executed either a certain number of times, or until a given runtime condition changes [36, p.219].

3.6 Explicit Concurrency

Two activities are concurrent if they are independent and can thus be interleaved in any order, or executed in parallel [36, p.219].[13] While declarative language constructs (see Section 3.4) and immutable state (see Section 3.2) can often be safely shared among concurrent activities, many languages also define explicit concurrency features. Interaction between concurrent activities can be supported in several different communication styles [42].

Unit A concurrency unit encapsulates a single explicitly-concurrent activity, programmed in either a declarative or imperative style, together with any necessary mutable state (see Section 3.3).

Message Passing In this communication style, concurrent activities communicate by exchanging messages, either synchronously (the activity waits until the message is received), asynchronously (the activity does not wait), or using a combination of modes [42]. A message can be defined as a data transfer, or as a request for some action to be carried out by the receiving activity [3].

Addressing A message may be addressed directly to a receiving activity; to a specific port on the receiving activity; or to an independent channel, which may have multiple receiving activities [36, CD p.263].

Sending A message-passing programming language can support any of three main message sending styles:

Asynchronous The sender waits only until the outgoing message has been copied to a safe location [36, CD p.268].

[13] The neutral term 'activity' is used here to mean any executable program fragment. The more usual terms 'process' and 'thread' are typically defined in terms of operating system concepts, such as memory management and scheduling, which are outside the scope of this paper.

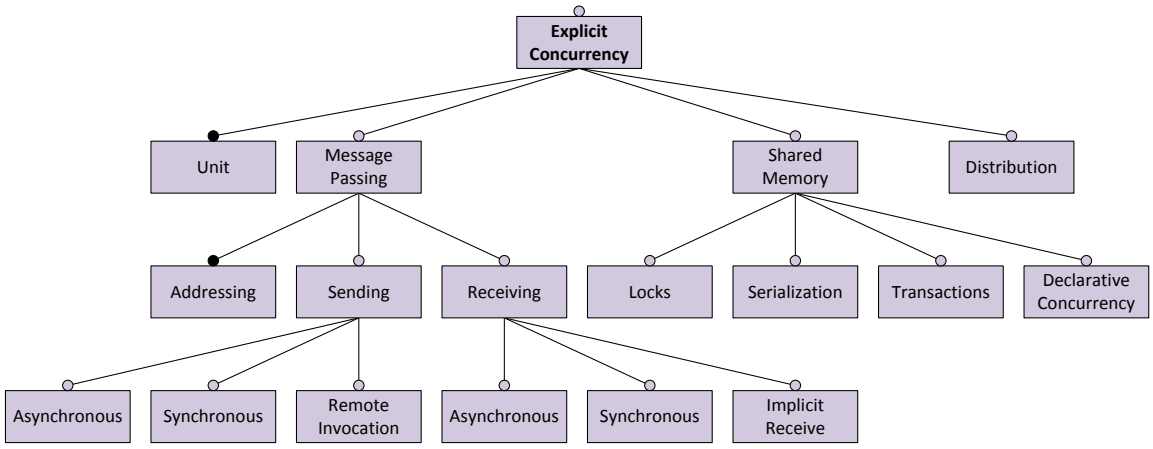

Figure 3.6: Explicit concurrency features.

	Erlang	Jason	Java
Unit	*Process*	*Agent*	*Thread*
Message Passing			
Addressing	Direct, by *pid* or registered name	Direct: `.send` or channel: `.broadcast`	
Sending			
Asynchronous	`Pid ! {c,2}`	`.send(Agents,Performative,c(2))` [b]	
Synchronous	✗	✗	✗ [a]
Remote Invocation	✗	`.send(Agents,askOne,c(X),Reply)` [c]	
Receiving			
Asynchronous	✗	`?c(X)[source(s)]` [d]	
Synchronous	`receive {c,X} -> do(X) end`	`.wait("+c(X)[source(s)]"); !do(X)`	
Implicit Receive	✗	Subject to *social acceptance* [7, p.71]	
Shared Memory			
Locks	✗	✗	*Synchronized* blocks
Serialization			*Synchronized* methods
Transactions			✗ [e]
Declarative Concurrency			`PipedInputStream`
Distribution	Named *nodes*	*SACI* or *JADE* [16]	*RMI* [f]

Table 3.6: Explicit concurrency in Erlang, Jason, and Java.

[a] In the object-oriented programming literature, some authors equate method invocation with message passing. We argue that Java method invocation (see Section 3.5) does not constitute a request for action (the receiving object cannot refuse) and in many cases does not involve a transfer of data (objects are passed by reference). Karmani et al. [24] list several third-party libraries which add some message-passing facilities to the Java language.

[b] Every Jason message must include a *performative*, which describes how the message content is to be interpreted. The available performatives are listed in Bordini et al. [7, p.118].

[c] The Jason `.send` and `.broadcast` *internal actions*, when used with the `askOne`, `askAll`, and `askHow` performatives, accept a variable parameter which is unified with the (first) message received in reply.

[d] Explicit message receipt in Jason is only possible with certain performatives, and relies on the *annotations* feature, which allows the agent to determine the source (`self`, `percept`, or another agent) of each of its beliefs.

[e] While Java has no general transactional programming features, the *atomic variable* classes provide some common operations (such as integer addition) with transactional characteristics.

[f] Java *Remote Method Invocation* (RMI) allows separate virtual machine instances to communicate by message passing with remote invocation. The high performance cost of virtual machine instances means RMI is not recommended for local concurrent activities.

Synchronous The sender waits until its message has been received [36, CD p.268].

Remote Invocation The sender waits until it has received a reply [36, CD p.268].[14]

[14] Though all three modes of message sending can be implemented in terms of the others, as distinct language features their syntax and performance characteristics can be separately optimised [36, CD p.271].

Receiving A message-passing activity can receive a message in any of three principal ways:

Asynchronous Also known as 'polling', asynchronous receive allows an activity to test if a message (possibly of a particular type) is available [36, CD p.272].

Synchronous The receiver waits until a message (possibly of a particular type) is received.

Implicit Subject to resource limitations, each received message implicitly triggers the creation of a new activity [36, CD p.272], which may parse the message parameters and carry out the requested actions.

Shared Memory In this communication style, concurrent activities communicate by operating on shared data structures.

Locks A lock or mutex is a low-level synchronization primitive that, once acquired by an activity, admits no further operations until it is released by the same activity [1, p.311].

Serialization Serialization allows a programmer to define "distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time" [1, p.304].

Transactions A transactional language offers computations with guaranteed atomicity (the intermediate steps are never visible to other activities) and isolation (once initiated, the result of the computation is unaffected by other activities) [32].

Declarative Concurrency Two concurrent activities may use streams [1, ch.3.5] (see Section 3.1), or shared single-assignment 'dataflow' variables (see Section 3.2), to communicate in a producer-consumer style [43, ch.4].

Distribution This feature allows concurrent activities to be distributed over two or more physically separate nodes.

3.7 Modularity

Modularity features allow a system to be divided into "coherent parts that can be separately developed and maintained" [1, p.217], and then optionally reused, both internally and externally, for economic gain [19]. Baldwin and Clark [6] propose a general theory of modularity, based on six operators which concisely describe the possible evolutionary paths for a modular structure.[15] For software systems, two of these - splitting and substitution - appear to require explicit support at the programming language level.[16]

Unit A module is a unit to which a responsibility is assigned; it consists of both data structures and the procedures which access and modify them [31].

Description Separating module descriptions from their implementations allows modules to specify the services they require, without explicitly naming the modules that provide those services.

Encapsulation The general purpose of a module is to "hide some design decision from the rest of the system" [31]. Some languages provide encapsulation features to enforce this hiding of information.

Splitting Features in this category support the splitting of a monolithic software design, or an existing software module, into separate modules.

Extension Extension features allow a programmer to create a new module by adding functionality to an existing module [36, p.468]. The namespaces of both modules are combined in the new module; a language offering this feature must define rules to resolve any name collisions.

Composition Composition allows a programmer to encapsulate one or more existing modules within a new module [43, p.411]. Each module retains a separate namespace; the new module mediates access to and between the modules it encloses.

Substitution Re-implementing a module is a common activity in software engineering. The alternative module may add functionality, remove existing functionality, repair errors, or implement the same functionality with different non-functional characteristics. Module substitution is a crucial step in the evolution of complex products [6, ch.5] and the development of software product families [39, p.736]. Compile-time substitution can be achieved using techniques which do not require language support, such as binary replacement [39, p.727]; this feature category therefore focuses on runtime module substitution features.

Replacement Runtime replacement consists of loading a new substitute module into a running system, and removing the replaced module, without restarting.

Selection Runtime module selection allows a program to choose between co-existing implementations of a module, depending on runtime conditions.

3.8 Metaprogramming

Metaprograms analyse, modify, and generate programs [44]. Since compiler construction and static analysis are outside the scope of this paper, this feature category is concerned specifically with reflective metaprogramming at runtime: programs that analyse, modify, and extend themselves. Metaprogramming features are categorised here according to the kinds of program artifact on which they operate.

Source Metaprograms manipulate plain text representations of their own source code.

Abstract Syntax Metaprograms read and modify their own partially-compiled source code, which is represented using the data structuring facilities of the programming language.

Binary Metaprograms operate on their own compiled binaries.

3.9 Input/Output

This feature category is concerned with input from, and output to, human users and 'environments'.[17] An environment mediates access to resources [45] (both hardware and software) and provides the "conditions" under which actors, agents, or objects exist [30].[18]

[15] An alternative theoretical treatment of modularity is given by Bracha and Lindstrom [8]. In this approach, the modularity features of real programming languages can be formally described as combinations of six low-level module manipulation operators: merge, restrict, project, select, override, and rename. A comparison of the modularity features of Erlang, Jason, and Java in terms of these operators is left for future work.

[16] The other four modular operators are augmenting (adding a new module to an existing system), excluding (removing a module from an existing system), inversion (making hidden functionality explicitly available as a module), and porting (moving a module to another system) [6, ch.5]. In practical software development, augmenting and excluding are easily supported either by simple condition flags, or by substituting null implementations [39, p.734]. Inversion can usually be implemented by splitting; and though porting has historically been important in language design, it is typically handled in modern languages at the virtual machine level.

[17] With the exception of the agent programming community, the subject of input/output receives relatively little attention in the programming language concepts literature. Considering the practical need for modern languages to integrate smoothly with large databases, Internet services, and a wide range of peripheral devices, we find this surprising. Consequently, input/output is included here as a fully-fledged feature category.

[18] Many agent programming researchers would define an environment more strongly, to include the requirement that the environment should mediate "interaction among agents" [45]. However, this definition does not generalise easily to other programming paradigms, as it seems to preclude direct

	Erlang	Jason	Java
Unit	*Module*	*Agent*	*Class*, *package*
Description	Custom *behaviours*	✗	*Interfaces*
Encapsulation	Private unless *exported*	✗ [a]	*Access modifiers*: `public`, `private`
Splitting			
Extension	`-include("other.hrl")`	`{include("other.asl")}`	Single inheritance
Composition	✗	✗	Classes only, using private fields
Substitution			
Replacement	On fully qualified function call [b]	`.kill_agent` and `.create_agent`	With custom `ClassLoader` [c]
Selection	Modules are first-class constructs	With *SACI* or *JADE* directories [d]	By polymorphism

Table 3.7: Modularity in Erlang, Jason, and Java.

[a] By default, the implicit receive feature (see Section 3.6) of the Jason interpreter allows an agent to read and modify the belief, goal, and plan base of any other. This behaviour can be changed by customising the interpreter's `socAcc` method [7, p.146].

[b] An Erlang function call of the form `module:function()` causes `module` to be replaced. *Current* and *old* versions of a module can be active simultaneously; an attempt to load a third version causes the old code to be purged, and any processes still running it to be terminated.

[c] Java custom class loaders allow classes, but not objects, to be directly replaced at runtime. To achieve runtime replacement of an object, the application must explicitly re-instantiate that object and discard the old instance for garbage collection.

[d] The SACI and JADE platforms provide *yellow pages* directories, in the form of *directory facilitator* agents, which can be used to select at runtime between multiple application agents offering the same services. However the Jason language itself offers no direct support for service description.

	Erlang	Jason	Java
Source	Write-only `erl_scan`, `erl_parse` modules	*Plan library manipulation*: `.add_plan(P)`	✗ [a]
Abstract Syntax	Write-only `compile` module	✗	Read-only *reflection* API
Binary	✗	✗	✗ [b]

Table 3.8: Metaprogramming in Erlang, Jason, and Java.

[a] Java source code can be compiled at runtime with the `javax.tools.JavaCompiler` library, however inspection and modification of currently-executing source code is not explicitly supported.

[b] Java class files (but not objects) can be read and modified at runtime using external libraries such as ASM or the Apache Byte Code Engineering Library. Limited runtime modification of classes and objects is also provided by the *instrumentation* API.

	Erlang	Jason	Java
Interactive	*Shell* read-eval-print loop	✗	*Swing* GUI library [a]
Message Passing			
Commands	In C and C++; *port drivers* [b]	Java *environment actions*	In C, C++, and Assembly; *JNI* [c]
Asynchronous	With `driver_async` C function	✗	✗
Synchronous	Using `port_command`	`go(left)`	`go("left")` [d]
Active Sensing	With `driver_output` C function	✗ [e]	`String msg = prompt("?")`
Events	As messages to the *port owner*	Individualised *percepts*	JNI *callbacks* to Java
Asynchronous	✗	`?at(X,Y)[source(percept)]`	With JNI field access
Synchronous	Using `receive`	`.wait("+at(X,Y)[source(percept)]")`	✗
Handlers	✗	`+at(X,Y)[source(percept)] <- do(X)`	With JNI method calling
Shared Data			
Streams	`file` and `io` modules	✗	`FileInputStream`, `FileReader`
Databases	`odbc` module	✗	*JDBC*

Table 3.9: Input/output in Erlang, Jason, and Java.

[a] Many additional interaction features for Java, such as the SWT graphical user interface (GUI) library, are provided by third-parties.

[b] Erlang also provides several other message passing input/output mechanisms, including *C nodes*, a Java nodes library called *jinterface*, TCP/IP and UDP sockets, and the newly-developed *Natively Implemented Functions (NIFs)*.

[c] JNI is the Java Native Interface.

[d] Before use, a Java native method must be explicitly loaded with `System.loadLibrary` and declared using the `native` keyword.

[e] While a Jason environment action cannot return a data item to its caller, it may directly change the percepts of any agent [7, p.106].

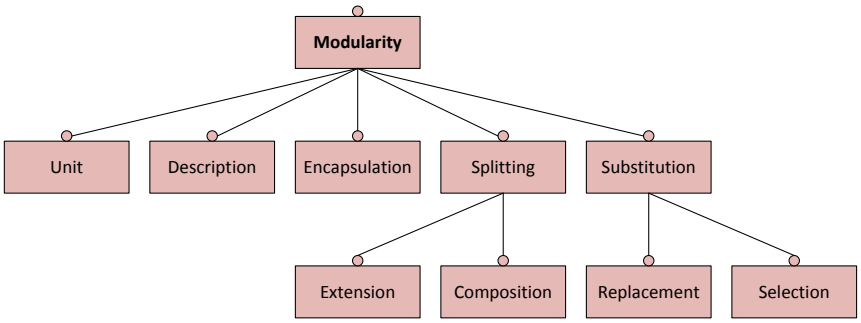

Figure 3.7: Modularity features.

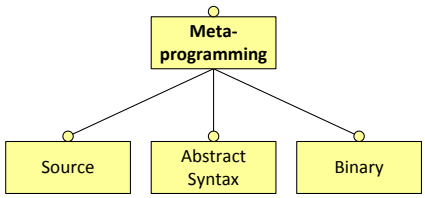

Figure 3.8: Metaprogramming features.

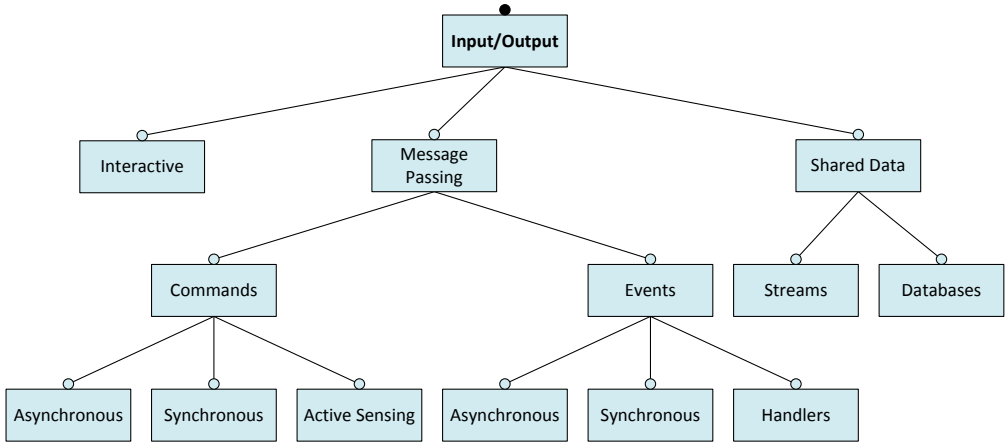

Figure 3.9: Input/output features.

The environment may consist of many concurrent activities, but unlike the concurrent activities discussed in Section 3.6, the environment is often defined in a different (usually lower-level) programming language.

Interactive A language may offer dedicated facilities for interaction with human users, in addition to those provided by the environment.

Message Passing In this mode of input/output, an activity communicates with its environment by sending commands and responding to events. As in Section 3.6, a message can be defined as a data transfer, or as a request for some action to be carried out [3]. In practice, message passing mechanisms for input/output are often different from those used in peer communication, because the environment is potentially unbounded and its structure may be unknown.

Commands Commands allow the programmer to request the environment to perform an action, which may either be predefined or specified in an intermediate command (or 'shell') language.

Asynchronous The command and its parameters (if any) are copied to a buffer, and the invoking activity resumes immediately.

Synchronous The invoking activity is suspended until the action begins.

Active Sensing The invoking activity is suspended until the action is complete. In this mode, the action may return a data item describing the result, or a simple success/failure indicator.

Events The environment may notify the activity of changes using events. The programmer may be required to explicitly subscribe to events of interest; alternatively, all environment events are made available to every activity. Events may be cached by the environment and retrieved individually by the activity, or matched against a list of event handlers and discarded. An activity can respond to an environment event in one of three ways:

Asynchronous The activity tests to see if a specific environment event is available.

Synchronous The activity waits until a specific environment event occurs.

Handlers The event implicitly triggers the execution of an event handling procedure, which may depend on the event type and content. The event handler may execute as an independent activity.

Shared Data In analogy with communication between concurrent activities by shared memory, communication with the environment may be achieved by the access and modification of shared data structures.

Streams Streams (see Section 3.1) can be used for input from, and output to, potentially unbounded data sources and sinks,

communication by shared memory between objects. We argue that this interaction mediation should be considered an optional property.

such as files, network connections, and character terminals. In this input/output mode, only one 'end' of the stream is visible to the application programmer, for either input or output. A language may define stream types for binary data, text, or structured records.

Databases A database is a shared collection of logically related data, with a self-describing structure. Access to a database is usually controlled by a database management system, which ensures the security and integrity of the data in the presence of concurrent access [9].

4. Conclusions

This paper proposes and validates a feature model of actor, agent, and object programming languages. The feature model allows comparison across a wide range of previously disparate programming language styles, and is designed to be extensible. The three language mappings used to validate the model should also be helpful to practitioners in deciding whether to use Erlang, Jason, or Java for a particular development project.

4.1 Limitations of the Feature Model

Some important programming language concepts were not explicitly included in the current feature model. These concepts, which are not easily represented as atomic features, were excluded in order to maximize composability, as noted in Section 2.

Scope Van Roy and Haridi [43, p.507] define scope as the "part of the program text in which [a] member is visible". Scope is usually expressed in terms of specific language constructs, which makes comparison of scope rules between languages difficult.

Exceptions A runtime failure or exception is defined as an unexpected condition that cannot be handled locally [36, p.418]. The rules for exception definition, propagation, and recovery are necessarily dependent on the current context; like scope, failure is a cross-cutting concern that does not easily fit to a feature-based model.

Security Defined as "protection from both malicious computations and innocent (but buggy) computations", security is a global system property [43, p.208]. While certain languages have well-known security flaws (see, for example, Scott [36, p.353]), modern security mechanisms are typically implemented by the compiler, interpreter, virtual machine, or operating system.

Finally, the feature model does not include any value judgements on the presence or absence of language features. We argue that the value of a given feature is inherently application-dependent. A full-featured language will allow a wider range of programs to be concisely expressed, but at the cost of a more expensive implementation and a more challenging learning curve.

4.2 Future Work

The main research value of the presented feature model lies in the future work which it enables. Some of this work is outlined as follows.

- Mapping other actor, agent, and object programming languages to the feature model would allow it to be refined and validated further.
- The model could be extended to accommodate other language styles, such as procedural or pure-functional, and validated against example languages, such as C or Haskell.
- Analytical work is needed to explore the dependencies between features, and thus arrive at a more complete understanding of the actor, agent, and object programming languages design space. If two features F_1 and F_2 are truly independent (and therefore composable), it must be feasible to construct languages which have both F_1 and F_2, F_1 only, F_2 only, and neither F_1 nor F_2. Unidirectional and bidirectional dependencies between features are also possible.
- The model could be used as a basis for structured comparisons, including empirical comparisons, between programming languages in any of the actor, agent, and object styles. This work would require the development of objective criteria, to determine whether each feature is present or absent.

In the longer term, given a sufficient understanding of the application domains in which actor, agent, and object programming languages are commonly used, the values of the features in the model could be determined as functions of the domain characteristics. Approximations of these functions could perhaps be obtained by analysing feature usage in existing applications, by experimenting with toy problems, or by analysis. This knowledge of which features are desirable would then help a designer or practitioner to create or select a programming language which is appropriate to a particular application domain.

Acknowledgments

The authors would like to thank Tom Arbuckle, John Noll, Klaas-Jan Stol, Guy Wiener, and the anonymous reviewers, for providing many insightful comments on earlier drafts of this paper. This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero, http://www.lero.ie/ and by Siemens Corporate Technology CT T CEE.

References

[1] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. Mit Press, second edition, 1996. ISBN 9780262510875.

[2] G. A. Agha. *ACTORS - A Model of Concurrent Computation in Distributed Systems*. MIT Press series in artificial intelligence. MIT Press, 1990. ISBN 978-0-262-01092-4.

[3] D. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[4] J. Armstrong. A history of Erlang. In B. G. Ryder and B. Hailpern, editors, *HOPL*, pages 1–26. ACM, 2007.

[5] J. Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[6] C. Baldwin and K. Clark. *Design Rules: The Power of Modularity*, volume 1. The MIT Press, 2000.

[7] R. Bordini, J. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley-Interscience, 2007. ISBN 0470029005.

[8] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 282–290. IEEE, 1992.

[9] T. Connolly and C. Begg. *Database Systems: a Practical Approach to Design, Implementation, and Management*. Addison-Wesley Longman, 2005.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0201309777.

[11] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, pages 1–17. Citeseer, 2003.

[12] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[13] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 39–67. Springer, 2005. ISBN 0-387-24568-5.

[14] L. Dennis, B. Farwer, R. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for BDI languages. In *Proceedings of the Fifth International Conference on Programming Multi-Agent Systems (ProMAS 2007)*, pages 124–139. Springer-Verlag, 2007.

[15] L. A. Dennis, M. Fisher, and A. Hepple. Language constructs for multi-agent programming. In F. Sadri and K. Satoh, editors, *CLIMA VIII*, volume 5056 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2007. ISBN 978-3-540-88832-1.

[16] V. Fernández, F. Grimaldo, M. Lozano, and J. Orduna. Evaluating Jason for distributed crowd simulations. In *Proc. of the 2nd International Conference on Agents and Artificial Intelligence*, volume 2, pages 206–211, 2010.

[17] R. A. Finkel and S. N. Kamin. *Advanced Programming Language Design*. Addison-Wesley-Longman, 1996. ISBN 978-0-201-06824-5.

[18] M. Fisher. MetateM: The story so far. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *PROMAS*, volume 3862 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2005. ISBN 3-540-32616-2.

[19] W. Frakes and K. Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7): 529–536, 2005. ISSN 0098-5589.

[20] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. *Many-dimensional Modal Logics: Theory and Applications*. Elsevier, 2003.

[21] M. Hanus. Multi-paradigm declarative languages. In V. Dahl and I. Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007. ISBN 978-3-540-74608-9.

[22] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[23] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[24] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In B. Stephenson and C. W. Probst, editors, *PPPJ*, pages 11–20. ACM, 2009. ISBN 978-1-60558-598-7.

[25] C. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 23–34. ACM, 2008.

[26] T. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1996.

[27] N. Madden and B. Logan. Modularity and Compositionality in Jason. In *Proceedings of the Seventh International Workshop on Programming Multi-Agent Systems (ProMAS 2009)*, 2009.

[28] B. Martin, C. Pedersen, and J. Bedford-Roberts. An object-based taxonomy for distributed computing systems. *Computer*, 24(8):17–27, 1991.

[29] R. Meier and V. Cahill. Taxonomy of distributed event-based programming systems. *The Computer Journal*, 48(5):602–626, 2005.

[30] J. Odell, H. Van Dyke Parunak, M. Fleischer, and S. Brueckner. Modeling agents and their environment. In *Proceedings of the 3rd International Conference on Agent-oriented Software Engineering III*, pages 16–31. Springer-Verlag, 2002.

[31] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[32] S. Peyton-Jones. Beautiful concurrency. In A. Oram and G. Wilson, editors, *Beautiful Code*, pages 385–406. OReilly, 2007.

[33] S. L. Peyton-Jones and P. Wadler. Imperative functional programming. In *POPL*, pages 71–84, 1993.

[34] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *MAAMAW*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996. ISBN 3-540-60852-4.

[35] A. Schwarz, M. Mehta, N. A. Johnson, and W. W. Chin. Understanding frameworks and reviews: a commentary to assist us in moving our field forward by analyzing our past. *DATA BASE*, 38(3):29–50, 2007.

[36] M. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, third edition, 2009. ISBN 9780123745149.

[37] Y. Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1): 51–92, 1993.

[38] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2): 11–49, 2000.

[39] M. Svahnberg, J. Van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques. *Software: Practice and Experience*, 35(8):705–754, 2005. ISSN 1097-024X.

[40] L. Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[41] F. Turbak, D. Gifford, and M. Sheldon. *Design concepts in programming languages*. MIT Press, 2008.

[42] P. Van Roy. Programming paradigms for dummies: What every programmer should know. *New Computational Paradigms for Computer Music*, 2009.

[43] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN 9780262220699.

[44] E. Visser. Meta-programming with concrete object syntax. In D. S. Batory, C. Consel, and W. Taha, editors, *GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002. ISBN 3-540-44284-7.

[45] D. Weyns, A. Omicini, and J. Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-agent Systems*, 14(1):5–30, 2007.