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

H.R. Jordan, G. Botterweck, J.H. Noll, A. Butterfield, R.W. Collier

PII: S0167-6423(14)00050-1
DOI: 10.1016/j.scico.2014.02.009
Reference: SCICO 1711

To appear in: *Science of Computer Programming*

Received date: 9 March 2013
Revised date: 31 January 2014
Accepted date: 5 February 2014

Please cite this article in press as: H.R. Jordan et al., A feature model of actor, agent, functional, object, and procedural programming languages, *Science of Computer Programming* (2014), http://dx.doi.org/10.1016/j.scico.2014.02.009

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.

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

H.R. Jordan[a,*], G. Botterweck[a], J.H. Noll[a], A. Butterfield[b], R.W. Collier[c]

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

Abstract

The number of programming languages is large [1] and steadily increasing [2]. 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' [4], 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.

Keywords: programming languages, programming language constructs, actor model, agent-oriented programming, functional programming, object-oriented programming, feature modelling, C, Erlang, Haskell, Jason, Java

1. Introduction

Programming languages are traditionally viewed as belonging to particular paradigms, however the notion of a programming paradigm is imprecise [3, p.xiii]. Unlike scientific paradigms [5, p.148], programming paradigms 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 organize 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, in conjunction with this hoped-for empirical

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

evidence, the model 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 maximize 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.

3. Related Work

Feature-oriented domain analysis has only recently been applied to the comparison of programming languages [27]. Consequently this section first discusses some examples of feature-based surveys in related areas. Then an overview of the literature on comparisons and concepts of programming languages is presented.

3.1. Feature-Based Surveys

Martin et al. [28] studied 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. However, while some features are illustrated with examples, a complete mapping of a real distributed computing system to the taxonomy was not provided.

Krauter et al. [29] classified grid resource management systems for distributed computing according to 11 functional attributes. Each attribute is structured as a small separate sub-taxonomy, resembling a feature model. Mappings of 15 real resource management systems to this taxonomy demonstrate its applicability, and these mappings were used to identify feature combinations that have received comparatively little attention.

A taxonomic approach was also employed by Meier and Cahill [30] to survey the features of distributed event systems. Pseudocode examples or architectural diagrams illustrate each categorisation decision. Complete mappings of four contrasting systems demonstrate the taxonomy's wide applicability.

Czarnecki and Helsen [31] surveyed model transformation languages using a feature-based approach. As in this paper, feature modelling was used to survey and organize a domain, rather than a product family from a single vendor; however the scope of the survey also includes proposed and prototype approaches alongside industrial-strength implementations. Due to this wider scope, brief comments on individual languages were provided in place of comprehensive feature mappings.

3.2. Programming Language Comparisons

The Steelman programming language requirements [32] represent an early attempt to identify desirable features of programming languages in general. These requirements were developed collaboratively and iteratively [33], and are structured as a hierarchy. Some of the requirements represent objectively detectable features, such as boolean types; however, other requirements such as 'maintainability' and 'simplicity' are less well defined. By design, the Steelman requirements' scope only covers imperative languages. Mappings of Ada, C, C++, and Java to the Steelman requirements were later provided by Wheeler [34].

Shaw et al. [35] proposed a general method to compare programming languages based on the idea of a language 'core': a subset of features that gives the language its identity and captures its designer's intent. The method was used to compare Fortran, Jovial, Cobol, and the Ironman programming language requirements (a predecessor of the Steelman requirements discussed above). Unfortunately, the process of identifying each language's core appears to be partly subjective, as it is based on historical and philosophical considerations. The language cores are compared according to criteria such as control flow, efficiency, and decomposition of large systems.

Wichmann [36] compared Pascal and Ada from the programmer's perspective, with the objective of illustrating the advantages of Ada for major industrial projects. Facilities discussed include type equivalence of arrays, function overloading, exception handling, concurrency, and packaging. However, the comparison criteria are unstructured, and their definitions are specific to the target languages. In a similar style, Feuer and Gehani [37] compared Pascal with C based on features such as basic and structured types, symbolic constants, assignment and selection statements, and input/output. Unfortunately these features are also not structured into a reusable model.

Appelbe and Hansen [38] surveyed the Concurrent Pascal, Pascal Plus, Modula-2, Ada, Mesa, Edison, CLU, PLZ/SYS, and C systems programming languages using a feature-based technique. Part of the survey is organised around high-level feature groups, such as typing, sequential control, concurrency, encapsulation, input/output, and some attributes of modularity; within these groups, further lower-level features are defined. For every surveyed language, each feature is either present or absent, and in some cases examples of feature usage are given. The main drawback of this survey is that declarative languages and language features are not considered.

Belkhouche et al. [39] proposed an explicit, systematic method for programming language evaluation, and used this method to compare Ada and Modula-2. According to this method, the features of the evaluated language should first be enumerated, then each feature should be rated according to a set of seven evaluation criteria. Unfortunately, some of these criteria, such as 'consistency with commonly-used notations' and 'extensibility', are not well defined. The procedure to enumerate the language features is not described, and the example comparison used to illustrate the method is based only on type systems and concurrency.

Appeltauer et al. [40] surveyed context-oriented programming languages and language extensions. A set of core features for context-oriented programming was defined, and implementation approaches for these features were compared; features unique to particular languages were also listed. The information-hiding modularity features of Ada, C++, CLU, Eiffel, Fortran, Modula-2, Oberon, and Simula were compared by Calliss [41].

3.3. Programming Language Concepts

Detailed discussions of the general concepts of programming languages are found in several well-known texts [42, 43, 3, 44, 45]. A classic overview of some fundamentals is provided by Strachey [46], while a more recent perspective may be found in Van Roy [47]. In specific subject areas, Tratt [48] surveys type system concepts, and Gabbay et al. [49] present the theoretical foundations of logic programming. The concepts of concurrent logic programming are discussed in detail by Shapiro [50].

Dennis et al. [51, 52] 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 [53], AgentSpeak [13], and MetateM [54] languages, and leads to the identification and abstract specification of some 'missing' modularity features.

Hudak [55] introduces the key features of functional programming languages, and the lambda calculus. A more general theoretical treatment of declarative languages by Hanus [56] describes attempts to unify functional programming with logic languages and the constraint programming paradigm. Armstrong [57] employs an empirical method to discover the fundamental concepts (called 'quarks') of object-oriented programming, which are then surveyed.

4. Feature Model

Due to the different terminology used in the actor, agent, functional, object, and procedural programming 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 realises, with a concrete syntax and semantics, one of the abstract features described in this section.[1] 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. Each node represents either a feature or a *feature group* - a group of related or conceptually similar features. Features and feature groups are either mandatory (denoted by solid bubbles) or optional (denoted by hollow bubbles). A solid segment joining the connectors of two or more optional nodes indicates that at least one of those nodes must be present.

[1] A single language may offer multiple realisations 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.

Mappings from the five selected languages to the feature model are described in the accompanying tables. Lack of explicit support for a particular feature or a feature group is indicated by the '✗' symbol. If a feature is present, a `monospace` font is used to provide an example of that feature instance where space allows. A greyed-out cell indicates a feature group that is present, because one or more features within the group are present. *Italics* indicate language-specific terminology used in Kernighan and Ritchie [6], `http://www.erlang.org/doc`, `http://www.haskell.org/onlinereport/haskell2010`, Bordini et al. [11], and `http://download.oracle.com/javase/tutorial`. 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. The colours used in the text, tables, and feature diagrams carry no semantics; their only purpose is to assist in matching related sections, and thereby improve readability.

An overview of the feature model is given in Figure 1.

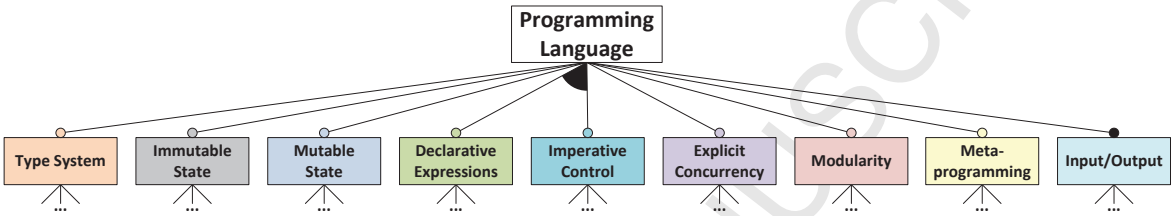

Figure 1: Overview of a feature model of actor, agent, functional, object, and procedural programming languages.

4.1. Type System

Type systems serve three related purposes in programming languages: to classify values, to determine their applicable operations, and to inform the compiler how much memory to allocate to store a value of the given type. In the first view, a type is "a constraint which defines the set of valid values which conform to it" [48]. In the second, types are abstract "specifications of functionality" [45, p.723], which define "legal usage contexts for the values they describe" [45, 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" [45, p.622].

Type Checking Type information may be checked for errors at compile time ('static' typing), at runtime ('dynamic' typing), or both [45, p.623].

Type 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 [45, ch.13].

Subtyping Subtyping allows two types to be compatible, without being the same [45, ch.12].

- **Safe Subtyping** 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 [48].

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

- **Booleans** Boolean types represent true or false.
- **Numbers** Number types represent numbers of specified precision, and can be signed or unsigned [44, p.294].
- **Characters** Character types represent letters in some encoding, such as ASCII or multilingual Unicode [44, p.295].

	C	Erlang	Haskell	Jason	Java
Type Checking	Compile time	Runtime [a]	Compile time	Runtime	Compile time [b]
Type Inference	✗	N/A	Compile time	N/A	✗
Subtyping	✗	✗	`class Eq a => Ord a`	✗	`class A extends B`
Safe Subtyping	✗		No casting		Casting with runtime checks
Base Types					
Booleans	✗[c]	✗	`Bool`	✗	`boolean`
Numbers	`char` `short`, `int`, `long`, `float`, `double`	*Integer*, *float*	`Integer`, `Int`, `Float`, `Double`	*number*	`byte`, `short`, `int`, `long`, `float`, `double`
Characters	`char`	ASCII code integer	Unicode, `Char`, `String`	Single-character *string* `"s"`	`char`
Enumerations	`enum`	✗	`data Compass = N | S | E | W`	✗	`enum Compass {N,S,E,W}`
Type Subranges	✗	✗	✗	✗	✗
Higher-order Types	✗	*Anonymous function*: `F=fun()->`	*"Curried" functions*: `f :: a -> b -> c` *Anonymous function*: `\ x -> ...`	*Plans* are strings: `P="+b <-`	✗[d]
Collections					
Tuples/Arrays	`int a[10]`	`T={a,b}`	`(a, b)` or `data (Ix a)=> Array a b = ...`	✗	`int[] a={4,2}` and *List* classes
Records/Dictionaries	`struct`	`-record(r,a,b)` [e]	`data D = R { f :: a, ...}`	See Table 3 [f]	`class D{int a,b;}` also *Map* classes
Lists	✗	`L=[a,b]`	`x:xs`, `[a,b]`	`L=[a,b]`	✗ but defined in *Collection* interface
Streams	POSIX `fmemopen()`	*Ports* for external I/O only	Infinite Lists, `xs where xs = x:xs`	✗	*InputStream*, *OutputStream*

Table 1: Type systems in C, Erlang, Haskell, 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`), based on type inference techniques.
[b] Some features of Java, such as *reflection*, require type checks to be delayed until runtime [45, p.624].
[c] No explicit boolean type; in boolean contexts, C coerces values to integers, where zero is treated as false, non-zero as true. Macros to simulate booleans were introduced in C99 [14].
[d] The Java *reflection* API provides a `Method` type for handling subroutines, however its use has many disadvantages and is officially discouraged.
[e] Erlang records are compiled as tuples; consequently, field names are replaced by integer indices at runtime. A single *dictionary* is also available to each Erlang process.
[f] Dictionaries can be built using *structures* in Jason, albeit with little support from the type system.

Enumerations Enumeration types represent ordered sets of named elements [44, p.297].

Type Subranges Subranges "compose a contiguous subset of the values of some discrete base type" [44, 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, methods, or procedures, allowing them to be "passed as parameters, returned by functions, or stored in variables" [44, p.290]. If higher-order types are true first-class language constructs, new values for those types can be computed at runtime [44, p.508] (see subsection 4.8).

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 [3, p.438].

Tuples/Arrays Tuples or arrays are indexed by integers.

Records/Dictionaries Records or dictionaries are indexed by any literal.

Lists Lists are unindexed, and of finite length.

Streams Streams are unindexed and unbounded; they are commonly used for input/output (see subsection 4.9) and concurrency (see subsection 4.6).

Figure 2 shows these concepts structured as a feature model, and Table 1 gives an overview of these features as they are implemented in the selected languages.

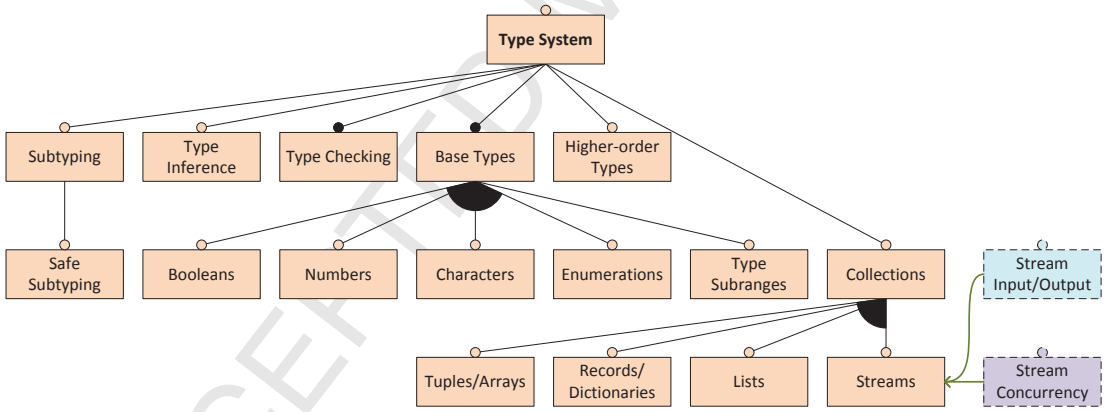

Figure 2: Type system features, including incoming dependencies from the Explicit Concurrency (see subsection 4.6, Figure 7) and Input/Output (see subsection 4.9, Figure 10) feature groups.

4.2. Immutable State

A crucial feature of many programming languages is "the possibility of associating values with symbols and later retrieving them" [42, 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.

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

Table 2: Immutable state in C, Erlang, Haskell, Jason, and Java.

[a] Constants in C can also be defined using macros: `#define C (2)`.

[b] Variation in Haskell arises in two ways: evaluating an expression which transforms its form, but not its value; and state-change that is hidden from view, accessed via a pointer (`IORef`) whose own value/address is immutable.

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

	C Static Allocation	C Dynamic Allocation	Erlang	Haskell	Jason	Java	Java *Map* Library Classes
State Cell Declaration	`int a`	`int *a = malloc(sizeof(int))`	✗	`IORef t`	✗	`int a`	✗ [a]
State Cell Assignment	By reference or value. [b] `a=2`	By reference or value. [b] `*a=2`	By value. `put(a,2)`	`a = newIORef 2`	By value. `+a(2)`	By reference or value. [c] `a=2`	By reference or value. `m.put("a",2)`
State Cell Valuedness	Single	Single	Single	Single	Multi. `+a(2); +a(3)`	Single	Single
State Cell Retrieval	`x=a`	`x=*a`	`get(a)`	`x <- readIORef a`	`?a(X)` [d]	`x=a`	`x=m.get("a")`
State Cell Modification	`a=3` [b]	`*a=3` [b]	`put(a,3)`	`writeIORef a 3`	`-a(2); +a(3)` or `-+a(3)` [e]	`a=3` [c]	`m.put("a",3)`
State Cell Deletion	✗	`free(a)`	`erase(a)`	✗ [f]	`-a(3)` or `-a(_)`	✗ [f]	`m.remove("a")`

Table 3: Mutable state in C, Erlang, Haskell, Jason, and Java.

[a] State cells in a Java *Map* are not declared individually, however the map itself must first be initialised: `Map<String,Integer> m = new HashMap<String,Integer>()`. Some `Map` implementations (including `HashMap`) permit null values; effectively, this allows names in the scope of a map to be declared before use.

[b] Assignment and modification of state cells in C is by value for primitive types (`int`, `double`, etc.). Arrays and structures are member copied on assignment and modification. Literal values can be multiply assigned to arrays on declaration (`int a[2] = {1,2}`), but subsequent multiple assignment or modification can only be achieved by copying memory locations, for example using `memcpy()`. Pointers are only shallow copied; afterwards, both source and target pointers will point to the same underlying data.

[c] Assignment and modification of state cells in Java is by value for primitive types (`int`, `double`, etc.) and by reference for instances of `Object`.

[d] The Jason interpreter attempts to match *test goals* against the agent's *beliefs* in reverse chronological order. The `include` preprocessor directive (see subsection 4.7) unfortunately interacts with this feature [58]. If no match is found, a test goal addition event is generated (see subsection 4.5).

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

[f] State cells in Haskell and Java are not explicitly deleted, but will be garbage collected.

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" [3, p.42].

Figure 3 shows these concepts structured as a feature model, and Table 2 gives an overview of these features as they are implemented in the selected languages.

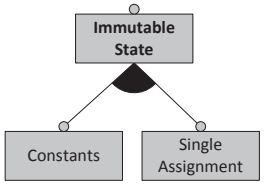

Figure 3: Immutable state features.

4.3. Mutable State

Van Roy and Haridi [3, 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 [3, p.410].

State Cell Declaration A declaration introduces a state cell name and indicates its scope [44, ch.3]. The declaration may also include type information (see subsection 4.1).

State Cell Assignment Assignment associates a state cell name 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) [44, p.225].

State Cell 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.

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

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

State Cell Deletion The mechanism by which a state cell, and its contents, are erased.

Figure 4 shows these concepts structured as a feature model, and Table 3 gives an overview of these features as they are implemented in the selected languages.

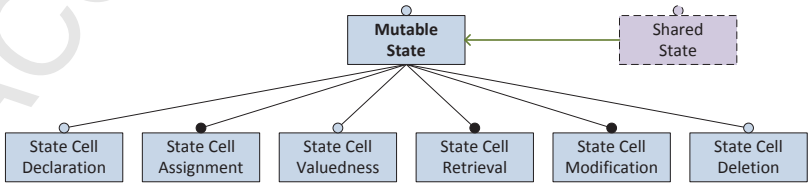

Figure 4: Mutable state features, including an incoming dependency from the Explicit Concurrency (see subsection 4.6, Figure 7) feature group.

	C	Erlang	Haskell	Jason	Java
Functions	✗	`add(A,B) -> A+B.`	`add a b = a + b`	✗	✗
Function Argument Evaluation		Applicative order	Lazy, with strictness annotations		
Declarative Conditional Expressions		`max(A,B) when A>B -> A; max(A,B) -> B.`[a]	`| x < 0 -> ... | x == 0 -> ...`[b]		
Tail Call Optimisation		Automatic when last expression is a function	Automatic		
Inference Rules		✗	✗	Horn clause *rules*. `positive(X) :- .number(X) & X>0.`	
Inference Rule Resolution				Top-down, left to right, depth first [c]	
Constraints		✗	✗	✗	

Table 4: Declarative expressions in C, Erlang, Haskell, Jason, and Java.

[a] In addition to the *guard sequences* illustrated here, Erlang also provides familiar *if* and *case* expressions.
[b] In addition to the definition guards shown here, Haskell has *if* and *case* expressions. Haskell conditional expressions may also use *pattern matching*.
[c] Jason rules are resolved with the agent's beliefs in reverse chronological order (see subsection 4.3).

	C	Erlang	Haskell	Jason	Java
Methods/Procedures	*Function*	✗	*IO/State Monad instances*	*Plan*	*Method*
Method/Procedure Invocation	By name or *function pointer*		Standard function call [a]	By *triggering event* and *context*. `+!te : ?c <-`	By name and matching parameter types
Method/Procedure Parameters	Positional, by value.		Positional, by value.	Positional, by value. `+!b(A,B)`	Positional, by reference or value.[b] `void s(int a,int b)`
Method/Procedure Return	Explicit, with termination. `return 2`		`return 2`, or last invocation.	✗	Explicit, with termination. `return 2`
Imperative Conditional Expressions	`if(x>y) {max=x;} else {max=y;}`, `max = x>y ? x : y`[c]		`if x > y then return x else return y`	`if(X>Y) {+max(X)} else {+max(Y)}`	`if(x>y) {max=x;} else {max=y;}`[c]
Iteration	`while`, `do while`, `for`		✗[d]	`while`, `for`	`while`, `do while`, `for`

Table 5: Imperative control in C, Erlang, Haskell, Jason, and Java.

[a] Haskell "procedures" are pure Haskell functions whose type belongs to a state mutation monad class, e.g. `IO`.
[b] Java method parameters are passed by value for primitive types (`int`, `double`, etc.) and by reference for instances of `Object`.
[c] C and Java also offer a `switch` statement, which selects between code fragments based on the value of a single variable, and supports *fall through*.
[d] Iteration constructs (e.g. `while`) can be defined in Haskell using recursion and the higher-order features.

4.4. Declarative Expressions

Finkel and Kamin [43] 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" [43, 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 [42, p.230]. This property is known as 'referential transparency' [55, p.362]. A function is distinct from a method or procedure (see subsection 4.5), in that side-effects are not permitted.

Function Argument 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) [42, p.16].[2] Under normal order evaluation, a function may return a value even if evaluation of some of its arguments would produce errors or not terminate [42, 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 [55, p.383]. The result of a computation under normal order or lazy evaluation may also depend on the order of function arguments [55, p.390].

Declarative Conditional Expressions Conditional expressions allow discontinuous functions to be defined, and are central to an 'equational reasoning' programming style [55, 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. A declarative conditional expression is referentially transparent.

Tail Call Optimisation 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 [45, p.1044]. A tail recursive language guarantees that recursive tail calls will be optimised to consume no additional memory resources, thus allowing iteration to be efficiently expressed. Tail call optimisation can only be activated in some languages using special syntax [42, p.35].

Inference Rules Inference rules allow new knowledge to be derived from existing facts[3]. 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.

Inference Rule Resolution Inference rule 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 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 [43, p.235]. The resolution process is potentially recursive, and therefore must be executed in a defined order over antecedent terms, to prevent accidental non-terminating queries [44, p.554]. Resolution is usually carried out left to right, and either depth first or breadth first.

[2] 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 [44, p.523]; if a particular argument is evaluated before the function body is entered, the function is strict in that argument [42, 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.

[3] 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 subsection 4.3). Efficient resolution in the presence of mutable state requires 'truth maintenance' mechanisms to ensure that previously-derived knowledge is only updated when necessary.

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.

Figure 5 shows these concepts structured as a feature model, and Table 4 gives an overview of these features as they are implemented in the selected languages.

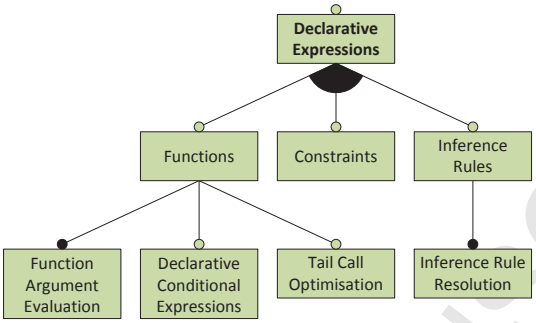

Figure 5: Declarative expression features.

4.5. 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 [44, p.220], and are also required to support structured input/output (see subsection 4.9) in declarative languages [59].

Methods/Procedures A method or procedure encapsulates a sequence of imperative control constructs, so they may be treated as a single unit [44, p.219]. A method or procedure is distinct from a function (see subsection 4.4): a method or procedure may change the state of the program (see subsection 4.3) or its environment (see subsection 4.9) via side effects.

- **Method/Procedure Invocation** The mechanism by which a method or procedure is selected for invocation.
- **Method/Procedure Parameters** A method or procedure 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 [44, 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) [44, p.395]; reference parameters may be used for output if the argument is mutable.
- **Return** Return values allow a method or procedure to send a result to its invocation context. The return mechanism may be implicit (the result is simply the value of the method or procedure 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 method or procedure [44, p.408].

Imperative Conditional Expressions Conditional expressions allow choices between two or more code fragments, depending on runtime conditions [44, p.219]. An imperative conditional expression is not necessarily referentially transparent; evaluating the condition may have side effects.

Iteration Iteration allows a code fragment to be executed either a certain number of times, or until a given runtime condition changes [44, p.219].

Figure 6 shows these concepts structured as a feature model, and Table 5 gives an overview of these features as they are implemented in the selected languages.

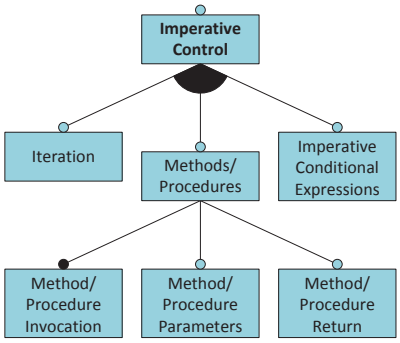

Figure 6: Imperative control features.

4.6. Explicit Concurrency

Two activities are concurrent if they can be interleaved in any order or executed in parallel, as determined by the underlying platform according to the number of CPU cores available.[4] While declarative programs (see subsection 4.4) can often be parallelized automatically by the compiler or interpreter, many languages also define features to explicitly support the concurrent execution of programs that rely on mutable state or imperative control. Interaction between explicitly concurrent activities can be supported in several different communication styles [47].

Concurrency 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 subsection 4.3).

Stream Concurrency In this communication style, two or more concurrent activities each use one end of a stream [42, ch.3.5] (see subsection 4.1) to communicate as producer and consumer [3, ch.4]. The use of a stream guarantees that data is sent and received in the same order.

Shared State In this communication style, concurrent activities communicate by modifying shared data structures.

- **State Cell Locking** A lock or mutex is a low-level synchronization primitive that, when applied to a state cell and acquired by an activity, admits no further operations on that state cell until the lock is released by the same activity [42, p.311].
- **Transactional State Cells** A transactional language offers computations on state cells 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) [62].

Code Locking Code locking, or 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" [42, p.304].

[4] 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.

	C	Erlang	Haskell	Jason	Java
Concurrency Unit	POSIX *Thread* [a]	*Process*	*Thread*	*Agent*	*Thread*
Stream Concurrency	✗ [b]	✗	*Control.Concurrent.Chan*: `getChanContents`, `writeList2Chan`	✗	`PipedInputStream`, `PipedOutputStream`
Shared State					
State Cell Locking	✗ [c]	✗	using `MVars`	✗	✗ [c]
Transactional State Cells	✗		STM [d]		*Concurrent collections* [e]
Code Locking	POSIX `pthread_mutex_lock()`, `pthread_mutex_trylock()`, `pthread_mutex_unlock()`; or using *semaphores*	✗	✗	✗	*Synchronized* methods and blocks, `wait()`, `notify()`, and `notifyAll()`

Table 6: Explicit concurrency part 1 - units, stream concurrency, shared state, and code locking, in C, Erlang, Haskell, Jason, and Java.

[a] The POSIX standard also defines C library functions and system calls for inter-*process* communication; as these require interaction with the operating system, they are presented in subsection 4.9.

[b] POSIX stream-oriented inter-process communication facilities (such as pipes and sockets) can be used between C threads, but they require interaction with the operating system; they are presented in subsection 4.9.

[c] In C and Java, every state cell is potentially accessible from every thread. Access to Java objects is controlled only by *access modifiers* (see subsection 4.7). In both languages, state cell locking can be achieved by disciplined use of code locking primitives; however, a rogue thread that does not observe the locking mechanism can still modify "locked" cells.

[d] Haskell supports Software Transactional Memory (STM) [60].

[e] The Java *atomic variable* classes also provide some common operations (such as integer addition) with transactional characteristics.

	C	Erlang	Haskell	Jason	Java
Message Passing Concurrency	✗[a]				✗[b]
Message Addressing		Direct, by *pid* or registered name	Via channels or `MVars`	Direct: `.send` or channel: `.broadcast`	
Message Sending					
Asynchronous Sending		`Pid ! {c,2}`	`putMVar`, `writeChan` [c]	`.send(Ags, Perf, c(2))` [d]	
Synchronous Sending		✗	✗[e]	✗	
Remote Invocation		✗	✗	`.send(Ags, askOne, c(X), Reply)` [f]	
Message Receiving					
Asynchronous Receiving		✗	`getMVar`, `readChan`	`?c(X)[source(s)]` [g]	
Synchronous Receiving		`receive {c,X} -> do(X) end`	✗[e]	`.wait("+c(X) [source(s)]"); !do(X)`	
Implicit Receiving		✗	✗	Subject to *social acceptance* [11, p.71]	

Table 7: Explicit concurrency part 2 - message passing concurrency in C, Erlang, Haskell, Jason, and Java.

[a] POSIX message-passing inter-process communication facilities (such as message queues) can be also be used between C threads, but they require interaction with the operating system; they are presented in subsection 4.9.

[b] In Java, method invocation (see subsection 4.5) alone is not sufficient to pass messages between threads; by default, an invoked ("receiving") method is always executed in the invoking ("sending") thread, regardless of where the method is located. Karmani et al. [61] list several third-party libraries which add message passing facilities to Java.

[c] The Haskell Chan type is an unbounded FIFO buffer.

[d] 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. [11, p.118].

[e] Facilities for synchronously sending and receiving messages in Haskell can be easily built on those for asynchronous message passing.

[f] 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.

[g] 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.

Message Passing Concurrency 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 [47]. A message can be defined as a data transfer between activities, or as a request by the sending activity for some action to be carried out by the receiving activity [57]. The order in which messages are sent is not necessarily preserved by the underlying platform.

Message 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 [44, CD p.263].

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

Asynchronous Sending The sender waits only until the outgoing message has been copied to a safe location [44, CD p.268].

Synchronous Sending The sender waits until its message has been received [44, CD p.268].

Remote Invocation The sender waits until it has received a reply [44, CD p.268].[5]

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

Asynchronous Receiving Also known as 'polling', asynchronous receiving allows an activity to test if a message (possibly of a particular type) is available [44, CD p.272].

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

Implicit Receiving Subject to resource limitations, each received message implicitly triggers the creation of a new activity [44, CD p.272], which may parse the message parameters and carry out the requested actions.

Figure 7 shows these concepts structured as a feature model; Table 6 and Table 7 give an overview of these features as they are implemented in the selected languages.

4.7. Modularity

Modularity features allow a system to be divided into "coherent parts that can be separately developed and maintained" [42, p.217], and then optionally reused, both internally and externally, for economic gain [64]. Baldwin and Clark [65] propose a general theory of modularity, based on six operators which concisely describe the possible evolutionary paths for a modular structure.[6] For software systems, two of these - splitting and substitution - appear to require explicit support at the programming language level.[7]

Modularity 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 [68].

[5] Though some 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 [44, CD p.271].

[6] An alternative theoretical treatment of modularity is given by Bracha and Lindstrom [66]. 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 C, Erlang, Haskell, Jason, and Java in terms of these operators is left for future work.

[7] 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) [65, ch.5]. In practical software development, augmenting and excluding are easily supported either by simple condition flags, or by substituting null implementations [67, 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.

	C	Erlang	Haskell	Jason	Java
Modularity Unit	File	*Module*	*Module*	*Agent*	*Class*, *package*
Module Description	*Header files*	Custom *behaviours*	✗	✗	*Interfaces*
Encapsulation	Public unless `static`	Private unless *exported*	All exported if no explicit `export`	✗ [a]	*Access modifiers*: `public`, `private`
Module Splitting					
Module Extension	`#include "mod.c"`	`-include("mod.hrl")`	✗	`{include("mod.asl")}`	Single inheritance
Module Composition	✗	✗		✗	Classes only, as private fields
Runtime Module Substitution			✗		
Runtime Module Replacement	`dlopen()`, `dlsym()`, `dlclose()`	On fully qualified function call [b]		`.kill_agent` and `.create_agent`	With custom `ClassLoader` [c]
Runtime Module Selection	With function pointers	Modules are first-class constructs		With *SACI* or *JADE* directories [d]	By polymorphism

Table 8: Modularity in C, Erlang, Haskell, Jason, and Java.

[a] By default, the implicit receive feature (see subsection 4.6) 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 [11, 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.

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

Table 9: Metaprogramming in C, Erlang, Haskell, Jason, and Java.

[a] C code can be compiled at runtime using 3rd-party libraries such as `libtcc` and `clang`, however the ability to manipulate currently-executing C code is platform dependent.

[b] Template Haskell can be used for *compile-time* metaprogramming [63].

[c] Java source code can be compiled at runtime with the `javax.tools.JavaCompiler` library, however manipulation of currently-executing source code is not explicitly supported. Java *annotations* can be used for *compile-time* metaprogramming.

[d] The `haskell-src` and `haskell-src-exts` packages give Haskell programmers access to parsing and compiler representations of programs.

[e] 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.

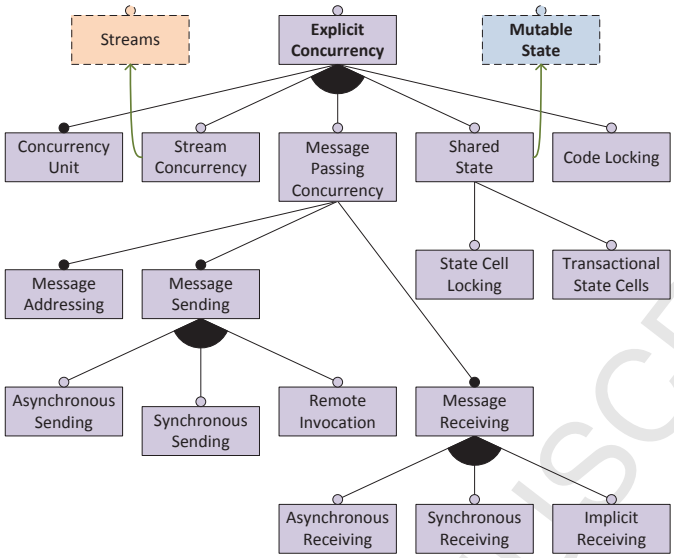

Figure 7: Explicit concurrency features, including outgoing dependencies on the Type System (see subsection 4.1, Figure 2) and Mutable State (see subsection 4.3, Figure 4) feature groups.

Module 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" [68]. Some languages provide encapsulation features to enforce this hiding of information.

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

- **Module Extension** Module extension features allow a programmer to create a new module by adding functionality to an existing module [44, 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.
- **Module Composition** Composition allows a programmer to encapsulate one or more existing modules within a new module [3, p.411]. Each module retains a separate namespace; the new module mediates access to and between the modules it encloses.

Runtime Module 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 [65, ch.5] and the development of software product families [67, p.736]. Compile-time substitution can be achieved using techniques which do not require language support, such as binary replacement [67, p.727]; this feature category therefore focuses on runtime module substitution features.

- **Runtime Module Replacement** Runtime module replacement consists of loading a new substitute module into a running system, and removing the replaced module, without restarting.
- **Runtime Module Selection** Runtime module selection allows a program to choose between co-existing implementations of a module, depending on runtime conditions.

Figure 8 shows these concepts structured as a feature model, and Table 8 gives an overview of these features as they are implemented in the selected languages.

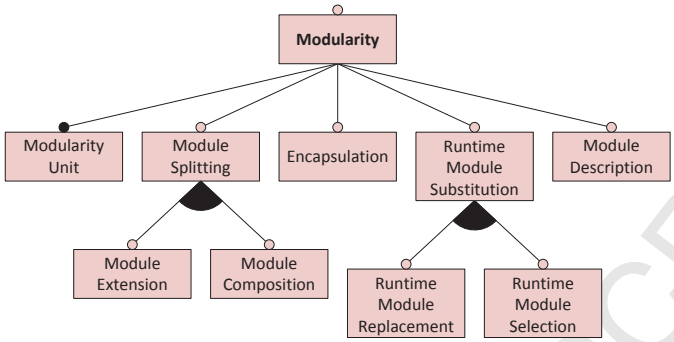

Figure 8: Modularity features.

4.8. Metaprogramming

Metaprograms analyse, modify, and generate programs [69]. 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 Metaprogramming Source metaprogramming allows programs to manipulate plain text representations of their own source code.

Abstract Syntax Metaprogramming Abstract syntax metaprogramming allows programs to read and modify their own partially-compiled source code, which is represented using the programming language's own data structuring facilities.

Binary Metaprogramming Binary metaprogramming allows programs to operate on their own compiled binaries.

Figure 9 shows these concepts structured as a feature model, and Table 9 gives an overview of these features as they are implemented in the selected languages.

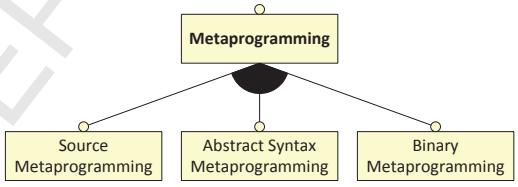

Figure 9: Metaprogramming features.

4.9. Input/Output

This feature category is concerned with input from, and output to, human users and 'environments'.[8] An environment mediates access to resources [70] (both hardware and software) and provides the "conditions"

[8] 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.

	C	Erlang	Haskell	Jason	Java
Stream Input/Output	*stdio*; POSIX `open()`, `read()`, `write()`, `close()` [a]	`file` and `io` modules	*System.IO* module [a]	✗	`System` streams, `java.io` and `java.nio` packages
Interactive Input/Output	✗ [b]	*Shell* read-eval-print loop	GHCi interpreter, various GUI libraries [c]	✗	*Swing* GUI library [d]
Shared Data	POSIX `mmap()`, `munmap()`; `shm_open()`, `shm_unlink()`		*System.Posix.SharedMem* module	✗	
Databases	POSIX *dbm* facility; database libraries [e]	`odbc` module	*Database.HDBC* module		*JDBC*

Table 10: Input/output part 1 - stream input/output, interactive input/output, and shared data in C, Erlang, Haskell, Jason, and Java.

[a] In C, the standard I/O facility or POSIX I/O system calls can be used to read from the environment on platforms that export environment data structures as files.
[b] There is no native C interpreter; numerous libaries (X, WinAPI, etc.) provide interactive GUI implementation.
[c] Haskell access to 3rd-party GUI libraries (e.g. Tcl/TK or wxWidgets) is via its Foreign Function Interface (FFI).
[d] Many additional interaction features for Java, such as the SWT graphical user interface (GUI) library, are provided by third-parties.
[e] There are numerous non-POSIX C libraries for database access, including ODBC.

	C	Erlang	Haskell	Jason	Java
Message Passing Input/Output					
System Calls	POSIX `mq_send()`, `mq_receive()`; `sendmsg()`, `recvmsg()`	In C and C++; *port drivers* [a]	*System* module	Java *environment actions*	`Runtime.exec()`, *JNI*, many libraries [b]
Asynchronous System Calls	POSIX non-blocking mode	With `driver_async` C function	Polling via `System.IO.hReady`	✗	`Process p = exec("cmd")`
Synchronous System Calls	POSIX blocking mode	Using `port_command`	*System.IO* module	`go(left)`	`go("left")` [c]
Active Sensing	POSIX `open() with O_SYNC flag` [d]	With `driver_output` C function	`hSetBuffering`	✗ [e]	`String msg = prompt("?")`
Event Notifications	POSIX *signals*	As messages to the *port owner*	*System.Posix.Signals* module	Individualised *percepts*	`WatchService` API, JNI *callbacks*
Asynchronous Notifications	POSIX `sigpending()`	✗	`getPendingSignals`	`?at(X,Y)[source(percept)]`	`WatchKey.poll()`
Synchronous Notifications	POSIX `pause()`, `sigsuspend()`, `sigwait()`	Using `receive`	`awaitSignal`	`.wait("+at(X,Y)[source(percept)]")`	`WatchKey.take()`
Event Handlers	POSIX `sigaction()`	✗	`installHandler`	`+at(X,Y)[source(percept)] <- do(X)`	Method calling with `JNIEnv`

Table 11: Input/output part 2 - message passing input/output in C, Erlang, Haskell, Jason, and Java.

[a] 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)*.

[b] The Java Native Interface (JNI) allows Java programs to interact with native code written in C, C++, and Assembly languages. Many other standard and third-party libraries support high-level interaction between Java programs and specific external systems.

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

[d] The `setvbuf()` facility can be used to make a C stream unbuffered, so that output appears at the destination as soon as it is written.

[e] While a Jason environment action cannot return a data item to its caller, it may directly change the percepts of any agent [11, p.106].

under which actors, agents, or objects exist [71].[9] The environment may consist of many concurrent activities, but unlike the concurrent activities discussed in subsection 4.6, the environment is often defined in a different (usually lower-level) programming language.

Stream Input/Output Streams (see subsection 4.1) can be used for input from, and output to, potentially unbounded data sources and sinks, 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.

Message Passing Input/Output In this mode of input/output, a program communicates with its environment by sending commands and responding to events. As in subsection 4.6, a message can be defined as a data transfer, or as a request for some action to be carried out [57]. 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.

System Calls System calls or 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 System Calls The system call and its parameters (if any) are copied to a buffer, and the invoking program resumes immediately.

Synchronous System Calls The invoking program is suspended until the action begins.

Active Sensing The invoking program 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.

Event Notifications An event notification mechanism allows the environment to notify the program of changes. The programmer may be required to subscribe to events of interest, or to selectively retrieve event notifications from a cache; alternatively, the environment may deliver event notifications directly by invoking event handlers provided by the program. A program can respond to an environment event in one of three ways:

Asynchronous Notifications The program tests to see if a specific event notification is available, and returns immediately.

Synchronous Notifications The program waits until a specific event notification occurs.

Event 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.

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

Shared Data In analogy with communication between concurrent activities by shared state, communication with the environment may be achieved by modifying data structures within the environment that are potentially accessible to multiple programs.

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 [72].

[9] Many agent programming researchers would define an environment more strongly, to include the requirement that the environment should mediate "interaction among agents" [70]. However, this definition does not generalise easily to other programming paradigms, as it seems to preclude direct communication by shared memory between objects. We argue that this interaction mediation should be considered an optional property.

Figure 10 shows these concepts structured as a feature model; Table 10 and Table 11 give an overview of these features as they are implemented in the selected languages.

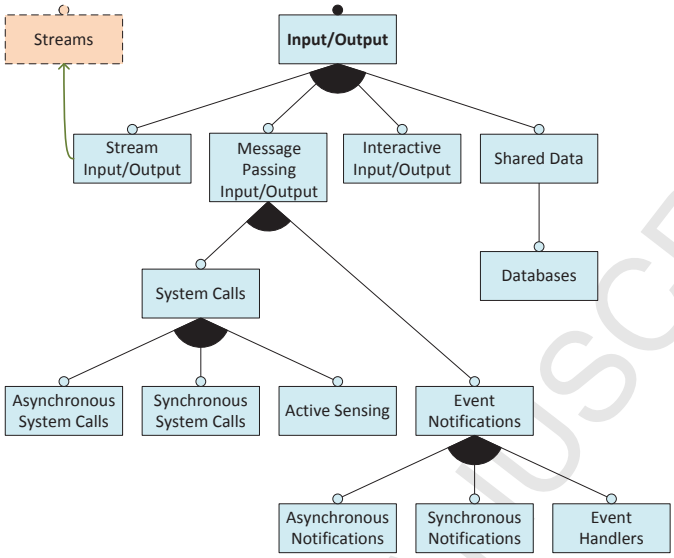

Figure 10: Input/output features, including an outgoing dependency on the Type System (see subsection 4.1, Figure 2) feature group.

5. Conclusions

This paper proposes and validates a feature model of actor, agent, functional, object, and procedural programming languages. The feature model allows comparison across a wide range of previously disparate programming language styles, and is designed to be extensible. The full feature model developed in section 4 is shown in figure Figure 11.

5.1. Limitations of the Feature Model

The five language mappings used to validate the model should be helpful to practitioners in assessing the suitability of C, Erlang, Haskell, Jason, and Java for a particular development project or software architecture. However, features are just one of many factors to take into consideration in such decisions. A key criterion is the availability of experts with sufficient experience in the chosen language. Other factors of potential importance include platform compatibility, development tool support, documentation, training materials, user community, and nonfunctional properties.

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 maximise composability, as noted in section 3.

Scope Van Roy and Haridi [3, 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 [44, 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 [3, p.208]. While certain languages have well-known security flaws (see, for example, Scott [44, p.353]), modern security mechanisms are typically implemented by the compiler, interpreter, virtual machine, or operating system.

Naming Languages such as Haskell offer sophisticated pattern-matching naming capabilities, however these are difficult to map to a feature model as they cut across multiple feature groups such as type systems and declarative expressions.

Most languages allow concurrent activities to be assigned to two or more separate nodes. However, many popular distributed computing mechanisms, such as CORBA, DCE/RPC, and JADE, are not part of their respective language specifications. Consequently, distribution topics have been excluded from this feature model.

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; "the simple presence of features is not a good indication of the worth of a language" [35]. 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.

5.2. 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 programming languages to the feature model would allow it to be refined and validated further.
- Analytical work is needed to further explore the dependencies between features, and thus arrive at a more complete understanding of the actor, agent, functional, object, and procedural 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.
- Feature models are commonly based on propositional logic; for instance, a feature f_1 may require a subfeature f_2. However, this feature model could be enriched with fuzzy relations between feature occurrences, to model both empirically-observed correlations and recommendations [73]. For example, whenever f_3 is present then f_4 may also be likely; or if f_5 is chosen, then f_6 is recommended.
- The model is designed to be used as a basis for structured comparisons, including empirical comparisons, between programming languages in any of the actor, agent, functional, object, and procedural 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, functional, object, and procedural 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 empirically, by studying feature usage in existing applications or by experimenting with toy problems. 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.

Acknowledgements

The authors would like to thank Marc-Philippe Huget, Tom Arbuckle, Klaas-Jan Stol, Guy Wiener, and the anonymous reviewers of AGERE! 2011 and Science of Computer Programming, for providing many insightful comments on earlier drafts of this paper. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (`http://www.lero.ie`) and by Siemens Corporate Technology CT T CEE.

[1] Rosetta code, 2007.
[2] R. Lämmel, C. Verhoef, Cracking the 500-language problem, pp. 78–88.
[3] P. Van Roy, S. Haridi, Concepts, Techniques, and Models of Computer Programming, MIT Press, 2004.
[4] J.-P. Rosen, What orientation should Ada objects take?, Communications of the ACM 35 (1992) 71–76.
[5] T. Kuhn, The Structure of Scientific Revolutions, University of Chicago Press, 1996.
[6] B. W. Kernighan, D. Ritchie, The C Programming Language, Second Edition, Prentice-Hall, 1988.
[7] J. Armstrong, Erlang, Commun. ACM 53 (2010) 68–75.
[8] J. Armstrong, A history of Erlang, in: B. G. Ryder, B. Hailpern (Eds.), HOPL, ACM, 2007, pp. 1–26.
[9] C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in: IJCAI, pp. 235–245.
[10] G. A. Agha, ACTORS - A Model of Concurrent Computation in Distributed Systems, MIT Press series in artificial intelligence, MIT Press, 1990.
[11] R. Bordini, J. Hübner, M. Wooldridge, Programming multi-agent systems in AgentSpeak using Jason, Wiley-Interscience, 2007.
[12] Y. Shoham, Agent-oriented programming, Artif. Intell. 60 (1993) 51–92.
[13] A. S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: W. V. de Velde, J. W. Perram (Eds.), MAAMAW, volume 1038 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 42–55.
[14] International Organization for Standardization, ISO/IEC 9899: Programming languages - C, 1999.
[15] International Organization for Standardization, ISO/IEC 9899: Programming languages - C, 2011.
[16] International Organization for Standardization, ISO/IEC 9899: Programming languages - C, 1990.
[17] Institute of Electrical and Electronics Engineers, IEEE 1003.1: Standard for information technology - portable operating system interface (POSIX) base specifications, issue 7, 2008.
[18] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[19] S. Apel, C. Kästner, An overview of feature-oriented software development, Journal of Object Technology (JOT) 8 (2009) 49–84.
[20] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report, Software Engineering Institute, 1990.
[21] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, The variability model of the Linux kernel, in: Proceedings of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 45–51.
[22] P. Clements, L. Northrop, Software product lines, Addison-Wesley, 2002.
[23] K. Pohl, G. Böckle, F. Van Der Linden, Software product line engineering, Springer, 2005.
[24] C. Kim, C. Kästner, D. Batory, On the modularity of feature interactions, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, ACM, pp. 23–34.
[25] A. Schwarz, M. Mehta, N. A. Johnson, W. W. Chin, Understanding frameworks and reviews: a commentary to assist us in moving our field forward by analyzing our past, DATA BASE 38 (2007) 29–50.
[26] J. Bosch, Design and use of software architectures: Adopting and evolving a product-line approach, Addison-Wesley Professional, 2000.
[27] H. Jordan, G. Botterweck, M. Huget, R. Collier, A feature model of actor, agent, and object programming languages, in: Proceedings of the Compilation of the Co-located Workshops on DSM, TMC, AGERE!, AOOPES, NEAT, & VMIL, ACM, pp. 147–158.
[28] B. Martin, C. Pedersen, J. Bedford-Roberts, An object-based taxonomy for distributed computing systems, Computer 24 (1991) 17–27.
[29] K. Krauter, R. Buyya, M. Maheswaran, A taxonomy and survey of grid resource management systems for distributed computing, Software: Practice and Experience 32 (2002) 135–164.
[30] R. Meier, V. Cahill, Taxonomy of distributed event-based programming systems, The Computer Journal 48 (2005) 602–626.
[31] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (2006) 621–646.
[32] Department of defense requirements for high order computer programming languages: Steelman, 1978.
[33] D. Fisher, DoD's common programming language effort, Computer 11 (1978) 24–33.
[34] D. Wheeler, Ada, C, C++, and Java vs. the Steelman, ACM SIGAda Ada Letters 17 (1997) 88–112.
[35] M. Shaw, G. Almes, J. Newcomer, B. Reid, W. Wulf, A comparison of programming languages for software engineering, Software: Practice and Experience 11 (1981) 1–52.
[36] B. Wichmann, A comparison of Pascal and Ada, The Computer Journal 25 (1982) 248–252.
[37] A. Feuer, N. Gehani, A comparison of the programming languages C and Pascal, ACM Computing Surveys 14 (1982) 73–92.
[38] W. Appelbe, K. Hansen, A survey of systems programming languages: Concepts and facilities, Software: Practice and Experience 15 (1985) 169–190.
[39] B. Belkhouche, L. Lawrence, M. Thadani, A methodical comparison of Ada and Modula-2, Journal of Pascal, Ada & Modula-2 7 (1988) 13–24.
[40] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, M. Perscheid, A comparison of context-oriented programming languages, in: International Workshop on Context-Oriented Programming, ACM.
[41] F. W. Calliss, A comparison of module constructs in programming languages, ACM SIGPLAN Notices 26 (1991) 38–46.
[42] H. Abelson, G. Sussman, J. Sussman, Structure and Interpretation of Computer Programs, Mit Press, second edition, 1996.
[43] R. A. Finkel, S. N. Kamin, Advanced Programming Language Design, Addison-Wesley-Longman, 1996.
[44] M. Scott, Programming Language Pragmatics, Morgan Kaufmann, third edition, 2009.
[45] F. Turbak, D. Gifford, M. Sheldon, Design concepts in programming languages, MIT Press, 2008.
[46] C. Strachey, Fundamental concepts in programming languages, Higher-Order and Symbolic Computation 13 (2000) 11–49.
[47] P. Van Roy, Programming paradigms for dummies: What every programmer should know, New Computational Paradigms for Computer Music (2009).
[48] L. Tratt, Dynamically typed languages, Advances in Computers 77 (2009) 149–184.
[49] D. Gabbay, A. Kurucz, F. Wolter, M. Zakharyaschev, Many-dimensional Modal Logics: Theory and Applications, Elsevier, 2003.
[50] E. Shapiro, The family of concurrent logic programming languages, ACM Computing Surveys (CSUR) 21 (1989) 413–510.
[51] L. Dennis, B. Farwer, R. Bordini, M. Fisher, M. Wooldridge, A common semantic basis for BDI languages, in: Proceedings of the Fifth

International Conference on Programming Multi-Agent Systems (ProMAS 2007), Springer-Verlag, pp. 124–139.
[52] L. A. Dennis, M. Fisher, A. Hepple, Language constructs for multi-agent programming, in: F. Sadri, K. Satoh (Eds.), CLIMA VIII, volume 5056 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 137–156.
[53] M. Dastani, M. B. van Riemsdijk, J.-J. C. Meyer, Programming multi-agent systems in 3APL, in: R. H. Bordini, M. Dastani, J. Dix, A. E. Fallah-Seghrouchni (Eds.), Multi-Agent Programming, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, Springer, 2005, pp. 39–67.
[54] M. Fisher, MetateM: The story so far, in: R. H. Bordini, M. Dastani, J. Dix, A. E. Fallah-Seghrouchni (Eds.), PROMAS, volume 3862 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 3–22.
[55] P. Hudak, Conception, evolution, and application of functional programming languages, ACM Comput. Surv. 21 (1989) 359–411.
[56] M. Hanus, Multi-paradigm declarative languages, in: V. Dahl, I. Niemelä (Eds.), ICLP, volume 4670 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 45–75.
[57] D. Armstrong, The quarks of object-oriented development, Communications of the ACM 49 (2006) 123–128.
[58] N. Madden, B. Logan, Modularity and Compositionality in Jason, in: Proceedings of the Seventh International Workshop on Programming Multi-Agent Systems (ProMAS 2009).
[59] S. L. Peyton-Jones, P. Wadler, Imperative functional programming, in: POPL, pp. 71–84.
[60] T. Harris, S. Marlow, S. L. P. Jones, M. Herlihy, Composable memory transactions, Commun. ACM 51 (2008) 91–100.
[61] R. K. Karmani, A. Shali, G. Agha, Actor frameworks for the JVM platform: a comparative analysis, in: B. Stephenson, C. W. Probst (Eds.), PPPJ, ACM, 2009, pp. 11–20.
[62] S. Peyton-Jones, Beautiful concurrency, in: A. Oram, G. Wilson (Eds.), Beautiful Code, O'Reilly, 2007, pp. 385–406.
[63] T. Sheard, S. Peyton Jones, Template metaprogramming for Haskell, in: M. M. T. Chakravarty (Ed.), ACM SIGPLAN Haskell Workshop 02, ACM Press, 2002, pp. 1–16.
[64] W. Frakes, K. Kang, Software Reuse Research: Status and Future, IEEE Transactions on Software Engineering 31 (2005) 529–536.
[65] C. Baldwin, K. Clark, Design Rules: The Power of Modularity, volume 1, The MIT Press, 2000.
[66] G. Bracha, G. Lindstrom, Modularity meets inheritance, in: Proceedings of the 1992 International Conference on Computer Languages, IEEE, pp. 282–290.
[67] M. Svahnberg, J. Van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software: Practice and Experience 35 (2005) 705–754.
[68] D. L. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (1972) 1053–1058.
[69] E. Visser, Meta-programming with concrete object syntax, in: D. S. Batory, C. Consel, W. Taha (Eds.), GPCE, volume 2487 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 299–315.
[70] D. Weyns, A. Omicini, J. Odell, Environment as a first class abstraction in multiagent systems, Autonomous Agents and Multi-agent Systems 14 (2007) 5–30.
[71] J. Odell, H. Van Dyke Parunak, M. Fleischer, S. Brueckner, Modeling agents and their environment, in: Proceedings of the 3rd International Conference on Agent-oriented Software Engineering III, Springer-Verlag, pp. 16–31.
[72] T. Connolly, C. Begg, Database Systems: a Practical Approach to Design, Implementation, and Management, Addison-Wesley Longman, 2005.
[73] K. Czarnecki, S. She, A. Wasowski, Sample spaces and feature models: There and back again, in: SPLC, IEEE Computer Society, 2008, pp. 22–31.

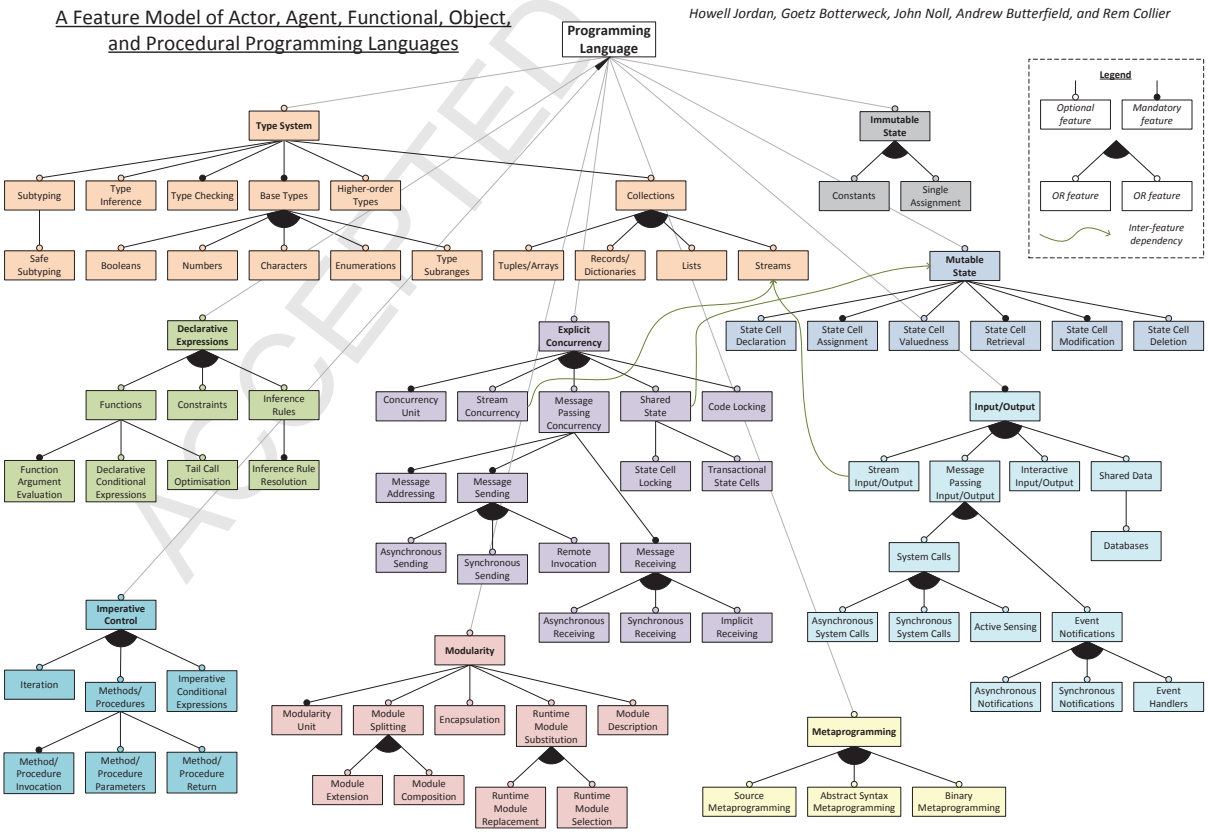

Figure 11: The resulting feature model of actor, agent, functional, object, and procedural programming languages – including all details.