Language and Architecture Paradigms as Object Classes: A Unified Approach Towards Multiparadigm Programming[⋆] [⋆⋆]

Diomidis Spinellis, Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ, UK

Abstract. Computer language paradigms offer linguistic abstractions and proof theories for expressing program implementations. Similarly, system architectures offer the hardware abstractions and quantitative theories applicable to the execution of compiled programs. Although the two entities are usually treated independently, object-oriented technology can be used to obtain a unifying framework. Specifically, inheritance can be used to model both programming languages as extensions to the assembly language executed by the target architecture, and system architectures as the root class of those paradigms. We describe how these principles can be used to model, structure and implement real multiparadigm systems in a portable and extendable way.

1 Motivation

Computer language paradigms offer linguistic abstractions and proof theories for expressing program implementations. Similarly, system architectures offer the hardware abstractions and quantitative theories applicable to the execution of compiled programs. It is widely accepted that each paradigm offers a different set of tradeoffs between efficiency, provability, elision, and implementation cost [17]. Thus, in the context of programming languages many believe that the *functional programming paradigm* is suited for applying formal proof theories on programs implemented in it, the *logic programming paradigm* is suited for expressing rule-based systems, and that efficient implementations are best expressed using the *imperative paradigm*. In the area of system architectures the imperative paradigm maps particularly well on the sequential von-Neumann architecture, while the referential transparency of the pure functional and logic languages can be advantageously used by parallel architectures.

⋆ Jürg Gutknecht Editor, *Programming Languages and System Architectures International Coneference*, pages 191–207, Zurich, Switzerland, March 1994. Springer Verlag. Lecture Notes in Computer Science 782.

⋆⋆ This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

Our research goal is the development and subsequent exploitation of a unifying framework for expressing the inter-relationship between system architectures and program paradigms. Such a framework could provide a solid basis for multiparadigm program development [30]. In particular, we are interested in multiparadigm programming environments, that allow a programmer to express a system using the most suitable paradigms. Absolute portability of the resulting implementations is not our goal as efficient implementations are closely tied to the underlying architecture. However, a relative amount of portability can be achieved by using the unifying framework and by abstracting suitable taxonomy characteristics of languages and architectures to identify the optimum coupling points between them.

2 Language and Architecture Paradigms as Classes

The word paradigm (from the Greek word $\pi\alpha\rho\acute{\alpha}\delta\epsilon\iota\gamma\mu\alpha$ which means example) is commonly used to refer to a category of entities sharing a common characteristic. Wittgenstein [29, p. 48] defines a paradigm by examining all the activities we call games. Among those activities there are some which possess some characteristic similarities equivalent to those exhibited by the members of a family. The *notion* "game" can only be defined by creating a list of all these typical cases that we call games, without being able to prescribe specific conditions for labelling an activity as a "game." In other words, we define games by listing some exemplar cases. In order to define an activity as a "game" it must share some common, but unspecified characteristics with those exhibited by the other members of the family; therefore the notion is only vaguely defined.

Kuhn used the notion of a *paradigm* in the scientific process by defining it as the scientist's view of the world and the structure of his or her assumptions and theories[3]. According to Kuhn [11, p. 10] a paradigm has a wider meaning than that of a scientific theory; it encompasses "law, theory, application and instrumentation together." Although Kuhn's examples are drawn from the history of physical science, his paradigm notion has been extended to a number of sciences [8, 6]. Paradigms are the basis of *normal science* which is related to all the activities of the established scientific tradition. Therefore, the formation of a paradigm is a sign of maturity for a given science.

In [24] it is suggested that as programming languages mature, attention is turning from languages to paradigms. An analogous statement can be made in relation to computer architectures where the characteristics of emerging technologies or quantitative theories are grouped into specific architecture paradigms. In trying to define the notion of a *programming* paradigm the most common definition found, is that of a "model or approach in solving a problem" [15], or the system architecture encompassing definition of "way of thinking about computer systems" [30]. A more general definition is given in [25, p. 21], where paradigms are described as rules for determining classes of languages according to some testable conditions. These conditions can be based on a number of abstraction criteria, such as the structure of a program or its state, or the development methodology. Similarly, system architecture classes can be grouped according to the cardinality of processing units, the implicit or explicit control of internal state, and the technologies available for mapping languages onto the given architecture.

[3] Our apologies to the many people who are offended by Kuhn's misuse of the word paradigm.

2.1 Paradigms as Object Classes

Implementation paradigms, whether at the hardware (system architectural) or software (programming language) level, are expressed using an appropriate notation. This notation can resemble the notation used by the machine that will execute the implementation (ranging from logic gate circuit diagrams, to state transition tables, to microcode listings) or it can resemble a more abstract notation suitable for describing implementations in that problem domain. At some point however, the implementation *will* be executed on a real machine and for this reason the semantic gap between the implementation paradigm and the programming paradigm of the target architecture must be bridged. This is usually done by an interpreter, a compiler or a hybrid technique. We regard all these methods as linguistic transforms from the paradigm notation to the target architecture notation. This view, although simple provides us with three insights:

1. A programming paradigm is nothing *magical*. All programming paradigms can be implemented on all architectures. Furthermore, in principle, there is no practical or theoretical reason for not being able to combine different paradigms, since they can all be mapped into the same concrete architecture.
2. The target architecture plays an important role when thinking of programming paradigms. The concept of the target architecture should be an integral part of multiparadigm systems and not an externally imposed specification, or an afterthought.
3. The target architecture naturally suggests a paradigm object hierarchy, with the target architecture forming the root of the hierarchy and other paradigms forming subclasses. Subclassing is used to create new paradigms, and inheritance to combine common features between paradigms.

Under the view outlined, most computer systems are intrinsically “multiparadigm”, since they embody a high-level description of some application implementation together with other lower-level elements, such as an operating system, or the underlying architecture microcode. From this point onwards, we will use the term “multiparadigm” to refer to some combination of implementation paradigms regardless of whether these are hardware or software based.

We found that the object metaphor suits the abstraction of a “programming or architecture paradigm”, and that by using it a common unifying model can be defined. In the following paragraphs we will examine how important aspects of object-oriented programming can be related to programming paradigms and multiparadigm programming. We will present the elements of the equation [23]:

object-oriented = objects + classes + inheritance

and in addition present the definition of class variables, instance variables and methods [14], in the context of programming and architecture paradigms.

In an object-based multiparadigm programming environment every paradigm forms a class, and every module written in that paradigm is an object member of that paradigm’s class. Paradigms form the class hierarchy with the target architecture being the root of it. Inheritance is used to bridge the semantic gaps between different paradigms.

Objects An object can be used as the abstraction mechanism for code written in a given programming paradigm or the realisation of a hardware architecture. Such objects have at least three *instance variables* (Fig. 1):

1. *Source code.* The source code contained in an object is the module code provided by the application programmer.
2. *Compiled code.* The compiled code is an internal representation of that specification (generated by the class *compilation method*) that is used by the class *execution method* in order to implement the specification.
3. *Module state.* The module state contains local data, dependent on the paradigm and its *execution method*, that is needed for executing the code of that object.

Every object has at least one *method*:

1. *Instance initialisation method.* The instance initialisation method is called once for every object instance when the object is loaded and before program execution begins. It can be used to initialise the module state variable.

As an example, given the imperative paradigm and its concrete realisation in the form of Modula-2 [28] programs, an object written in the imperative paradigm corresponds to a Modula-2 module. The *source code* variable of that object contains the source code of the module, the *compiled code* variable contains the compiled source, and the *module state* variable contains the values of the global variables. In addition, the *instance initialisation method* is the initialisation code found delimited between `BEGIN` and `END` in the module body. In an example closer to the system architecture, the *source code* variable represents the assembly listing of that compiled module, the *object code* variable represents the machine code, while the *module state* is contained in the processor's data memory allocated for that module.

Classes Collections of objects of the same paradigm are members of a class. All classes contain at least one class variable (Fig. 1):

1. *Class_state*: contains global data needed by the *execution method* for all instances of that class.

In addition, paradigm classes define at least four *methods*:

1. *Compilation method.* The compilation method is responsible for transforming, at compile-time, the source code written in that paradigm into the appropriate representation for execution at run-time.
2. *Class initialisation method.* The class initialisation method of a paradigm is called on system startup in order to initialise the class variables of that class. It also calls the instance initialisation method for all objects of that class.
3. *Execution method.* The execution method of a class provides the run-time support needed in order to implement a given paradigm.
4. *Documentation method.* The documentation method provides a description of the class functionality. It is used during the building phase of the multiparadigm environment, in order to create an organised and coherent documentation system.

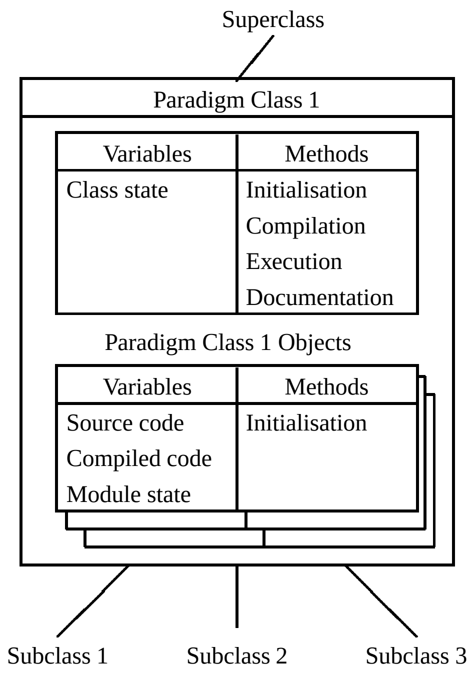

Fig. 1. Programming paradigm classes and objects

The compilation and execution methods also contain the machinery needed to implement the import and export call gates described in Sect. 2.3.

Taking as a paradigm class example the logic programming paradigm realised as Prolog compiled into Warren abstract machine instructions [22], the *class state* variable contains the heap, stack and trail needed by the abstract machine. In addition, the *compilation method* is the compiler translating Prolog clauses into abstract instructions, the class initialisation method is the code initialising the abstract machine interpreter, while the execution method is the interpreter itself. The above also holds if the abstract machine is realised as a concrete hardware architecture. In that case the execution method is the processor hardware, or — for microcode-based systems — its microcode.

Inheritance Inheritance is used to bridge the semantic gap between code written in a given paradigm and its execution on a concrete architecture. We regard the programming paradigm of the target architecture as the *root class*. If it is a uniprocessor architecture it has exactly one object instance, otherwise it has as many instances as the number of processors. The *execution method* is implemented by the processor hardware and the *class_state* is contained in the processor's registers. The *compiled code* and *module state* variables are kept in the processor's instruction and data memory respectively.

From the root class we build a hierarchy of paradigms based on their semantic and syntactic relationships. Each subclass inherits the *methods* of its parent class, and can thus use them to implement a more sophisticated paradigm. This is achieved, because each paradigm class creates a higher level of linguistic abstraction, which its subclasses can use.

As an example, most paradigms have a notion of dynamic memory; a class can be created to provide this feature for these paradigms. Two subclasses can be derived from that class, one for programmer-controlled memory allocation and deallocation and another for automatic garbage collection. As another example a simulation paradigm and a communicating sequential processes paradigm could both be subclasses of a coroutine-based paradigm. Subclassing is not only used for the run-time class execution methods. Syntactic (i.e. compile-time) features of paradigms can be captured with it as well. Many constraint logic languages share the syntax of Prolog, thus it is natural to think of a constraint logic paradigm as a subclass of the logic paradigm providing its own solver method, and extension to the Prolog syntax for specifying constraints. A paradigm class tree based around these examples is shown in Fig. 2. It is important to note that Fig. 2 only represents an example based on *one* possible set of abstraction characteristics. Other class hierarchies based on language attributes such as type system or block structure are possible and may be preferable.

2.2 Paradigm Inter-operation Design

Having described the basic structure of a multiparadigm system we must now deal with the problem of paradigm inter-operation. Languages supporting modularisation allow a problem to be decomposed in smaller problems. The entities representing the decomposed problem vary according to the language paradigm as summarised in the Table 1. Each one of them however is based on the basic computational model of input, computation, and output.

Table 1. Paradigms and their problem decomposition entities.

Paradigm	Decomposition entity
Imperative	Procedure
Functional	Function
CSP	Process
Logic	Predicate
Object-oriented	Method

When a decomposition entity is invoked (implicitly or explicitly), control is passed to it in conjunction with some input data. After the requisite computation is performed, control passes back to the invoking entity together with the resulting data. In some cases the computation entity may directly interact with input/output devices or modify its internal state. In those cases, input and/or output data does not have to be provided. Examples of explicit and implicit control transfer are given in Tables 2 and 3 respectively.

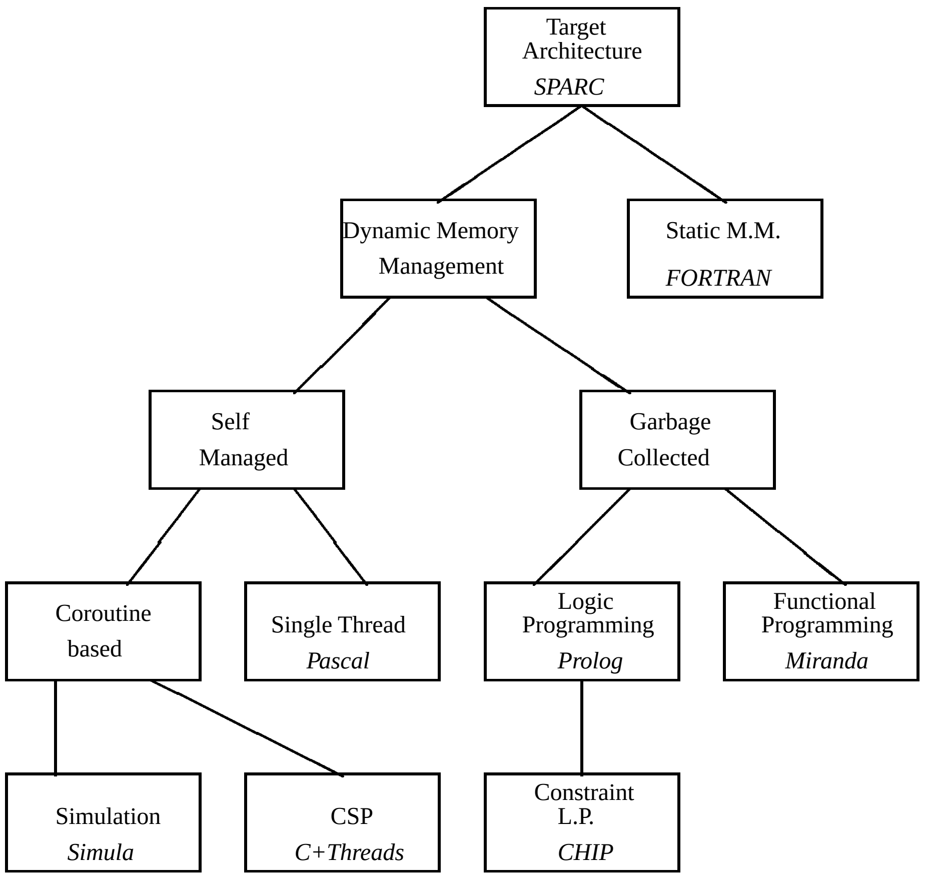

Fig. 2. Paradigm class tree structure example

Our approach supports explicit control transfer between paradigms, and implicit control transfer within a paradigm. In this way all semantic and implementation issues related to the implicit transfer of control are avoided at the expense of a less expressive system. The restricted version of control transfer is based on the encapsulation properties of the underlying objects: control transfer between the paradigms follows the control transfer conventions of their superclass. This recursive definition is followed until we reach the root class, the target architecture. Using the conventions of the parent paradigm ensures that no unexpected interactions occur. By unexpected interactions we mean implicit and unintended control transfer from one paradigm to the other. The subclasses are coded using the features and caveats of the parent class; this ensures that unexpected interactions will not occur. We will attempt to clarify this statement by four examples:

Non-Preemptive Von-Neumann Target Architecture The basic control transfer mechanism used is the explicit *procedure call*. Implicit calls between other paradigms

Table 2. Explicit control transfer in different paradigms.

Paradigm	Transfer mode
Imperative	Procedure call
Functional	Eager β reduction
Logic	Predicate invocation through its *call* port (c.f. Byrd model [2])

Table 3. Implicit control transfer in different paradigms.

Paradigm	Transfer mode
Imperative	Invocation of an interrupt handler
Functional	β reduction of a suspended expression in a lazy implementation
Logic	Predicate invocation through its *redo* port (backtracking)
CSP	A process is awaken as a result of an external event (e.g. an input or output buffer becomes full)

will be encapsulated within their respective classes and therefore the paradigms remain isolated.

Threads Subclass We assume now the existence of the *thread* programming subclass with primitives that allow the creation of multiple processes. Since this subclass was coded using the mechanisms of the parent paradigm, the threads created will be non-preemptive and therefore all implicit control transfers will happen *within* the threads paradigm. Therefore no implicit control transfers can occur between paradigms, other than subclasses of the *threads* paradigm. These subclasses will of course be coded to anticipate such control transfers.

Event-Driven Target Architecture In an event-driven target architecture, all paradigm compilers have to be able to cope with non-deterministic control transfers. For this reason the paradigms provided will — by definition — be able to cope with such transfers. A straightforward way to implement such a system would be to disable interrupts when paradigms that are unsuitable for such an environment are executed (a higher quality implementation would provide a better solution).

Parallel Target Architecture Here again the target architecture imposes some stringent requirements regarding synchronisation, memory sharing, message passing etc. All these requirements are inherited, and must be handled by all paradigm subclasses.

The data that is exchanged between the computation entities can be divided between data that is commonly available in similar representations[4] across many different paradigms, and data representations that are only supported in a limited number of paradigms. We say that a paradigm supports a given data representation if that form of

[4] Some paradigms may *cell* a particular representation in the form of a record usually containing its type and a pointer to the actual value.

representation can be

- expressed in the source code of that paradigm,
- dynamically created at run-time, and
- operated upon by the data manipulation primitives or libraries of that paradigm.

Examples of data representation commonly available across different paradigms and representations particular to specific paradigms are listed in Tables 4 and 5.

Table 4. Data representations commonly available across many paradigms.

Data type	Common representation
Integer value	Two, four or 8 bytes
String value	Character vector
Character	One byte (ASCII)
Floating point number	IEEE 488 byte sequence
Boolean	Single byte or bit
A finite predetermined collection of the above (record, structure, term)	Sequence of bytes representing the above

Table 5. Data representations particular to specific paradigms.

Data representation	Supporting paradigm
Uninstantiated value	logic programming
Curried function	functional programming
List of infinite length	functional programming
Pointer or reference	imperative programming
Valued data object whose value can be changed	imperative programming
Array	imperative programming

For the sake of simplicity our approach only supports inter-paradigm communication with data representations supported by both paradigms exchanging data. When the two paradigm implementations use dissimilar representations, it is straightforward to map one representation to the other. Examples of such transformations include the assembling or disassembling of plain values used in imperative paradigms to the cell-based representations common in declarative paradigms, or the changes between byte orderings for different representations of arithmetic types. This approach again trades limited expressive power for semantic and implementational simplicity.

2.3 Paradigm Inter-operation Implementation Abstraction

Paradigm inter-operation can be designed around an abstraction we name a *call gate*. A call gate is an interfacing point between two paradigms, one of which is a direct subclass of the other. We define two types of call gates, the import gate, and the export gate. In order for a paradigm to use a service provided by another paradigm (this could be a procedure, clause, function, rule, or a port, depending on the other paradigm), that service must pass thought its import gate. Conversely, on the other paradigm the same service must pass through its export gate. The call gates are design abstractions and not concrete implementation models. They can be implemented by the paradigm compiler, the runtime environment, the end user, or a mixture of the three. Each paradigm provides an import and export gate and documents the conventions used and expected. The input of the export gate, and the output of the import gate follow the conventions of the paradigm, while the output of the export gate, and the input of the import gate, follow the conventions of the paradigms' superclass. The target architecture paradigm combines its import and its export gate using the linked code as the sink for its export gate and the source for its import gate. Call gates can make the paradigm inter-operation transparent to the application programmer, and provide global scale inter-operation using only local information.

Figure 3 illustrates an example case. Assume that a module written in *paradigm 2* is using a facility implemented in *paradigm 1.1*. The module written in *paradigm 1.1* will export that facility (using the syntax and semantics appropriate to *paradigm 1.1*) to its superclass (*paradigm 1*) through its export gate, thus converting it to the data types and calling conventions used by paradigm 1. *Paradigm 1* will again pass it through its export gate, converting it to the conventions used by *paradigm 0*, the target architecture. (For example, the calling conventions of the Unix system can include the passing of parameters through a stack frame, and the naming of identifiers with a prepended underscore.) In this form the facility will again be imported from the pool of linked code by *paradigm 1* and made available to its subclasses using its conventions. The facility can then be imported and used by *paradigm 2* which can understand the calling conventions of *paradigm 1*. Although during the path described the facility crossed three paradigm boundaries, in all cases the paradigm just needed to be able to map between its calling conventions and data types and those of its superclass.

We must note at this point that the class hierarchy is not visible to the application programmer. The hierarchy is useful for the multiparadigm programming environment implementor, as it provides a structure for building the system, but is irrelevant to the application programmer, who only looks for the most suited paradigm to build his application. This is consistent with the recent trend in object-oriented programming of regarding inheritance as a *producer's mechanism* [13], that has little to do with the end-user's use of the classes [4].

3 An Exemplar Multiparadigm Programming Environment

3.1 Design Objectives

Blueprint is an exemplar multiparadigm programming environment, built using the

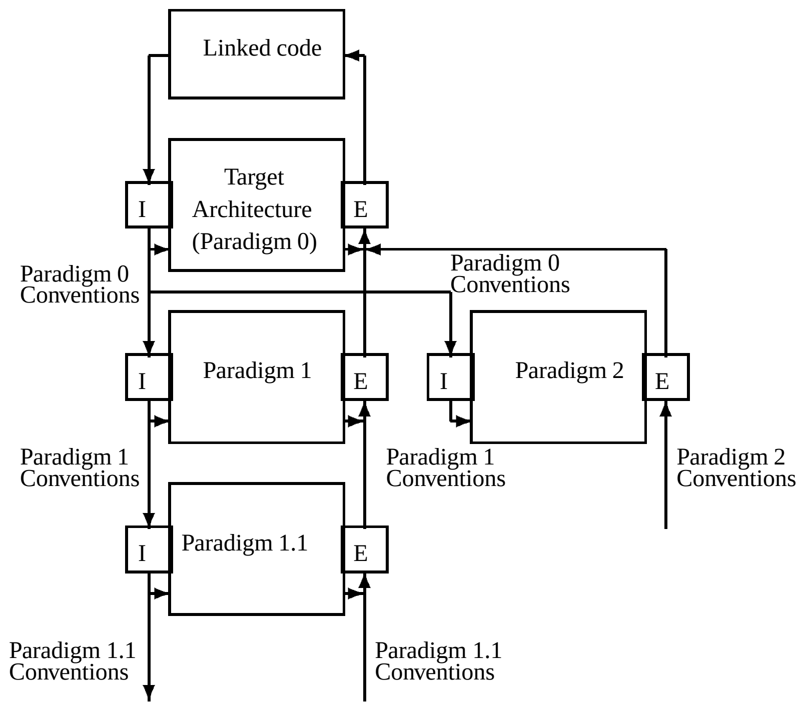

Fig. 3. Paradigm inter-operation using call gates

object-based approach. It was implemented in order to prove the viability of the object-based approach to multiparadigm programming and its design was centred around the following objectives:

– realisation of a wide variety of diverse programming paradigms, and implementation methods,
– provision of a non-trivial class hierarchy, including the abstraction of common characteristics in a special superclass,
– incorporation of existing tools, and
– ability to bootstrap the system and implement a non-trivial application in order to test and use it as much as possible.

The *blueprint* name is derived from the acrostical spelling of the paradigms provided, namely:

– **B**NF grammar descriptions (*bnf*),
– **l**azy higher order functions (*fun*),
– **u**nification and backtracking (*btrack*),
– **r**egular expressions (*regex*),
– **i**mperative constructs (*imper*) and,

– **t**erm handling (*term*).

All paradigms are provided in the form of individual paradigm compilers: tools that convert the code expressed in a given paradigm into object code that can be linked and executed together with code from other paradigms.

3.2 System Structure

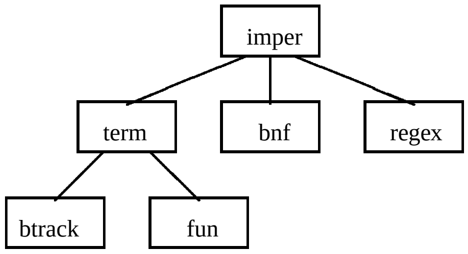

Fig. 4. *Blueprint* class hierarchy

The target paradigm of *blueprint* is the imperative paradigm provided by the target architecture, which in our case is that of the Sun SPARC computer. The class structure of the paradigm classes implemented can be seen in Fig. 4. Term expressions are the natural data objects, for both functional and logic languages; the provision of the *term* class is based on this observation and, in addition, provides a practical vehicle for their implementation.

It is important to note that the tree structure is only used in order to design and implement the system. The structure is transparent to a programmer using *blueprint* who is presented with a flat structure of all the paradigms (Fig. 5). In the following paragraphs we briefly describe each *blueprint* paradigm.

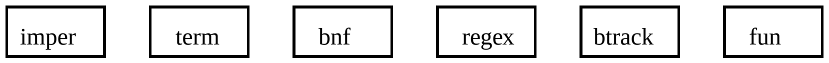

Fig. 5. Programmer's view of *blueprint*

Imperative Paradigm *Imper*, the imperative paradigm, is provided in the form of the C programming language. It is the one closest to the target architecture, and the calling and naming conventions of the language are used as a common interface for the other paradigms.

BNF Grammar and Regular Expression Paradigms The *bnf* (BNF-grammar) and *regex* (regular expression) paradigms are used to encapsulate *yacc* [10] grammar descriptions and *lex* [12] lexical analyser specifications, as objects. The main advantage of this encapsulation is the ability to use more than a single grammar description or lexical analyser specification within the same project. This is achieved by "protecting" the global variable and function names that *yacc* and *lex* define by prepending them their object (module) name. Both paradigms were implemented by using special tools to create multiparadigm environment conformant compilers out of the standard Unix *yacc* and *lex* generators. Inter-operation with the imperative paradigm is achieved by using the standard *lex* and *yacc* interfacing conventions, as modified by the object encapsulation scheme.

Rule-rewrite Paradigm *Term*, the term-based rule rewrite paradigm abstracts the notion of a *term* used by both the functional and logic programming paradigms. Its syntax resembles that of Prolog, but it uses a deterministic rule-rewrite execution model with predefined argument mode declarations, resembling the functionality provided by Strand [7]. *Term* is implemented in *term*, *imper*, *bnf*, and *regex* as a compiler that translates *term* into C. It was bootstrapped using the SB-Prolog compiler [5], and a semi-automatic translation process. Inter-operation with the imperative paradigm is provided by documenting the compiled form of the *term* "predicates" and providing access and constructor functions for the term abstract data type, in its converted form of C *structures*.

Logic Programming Paradigm *Btrack*, the logic programming paradigm provides the backtracking execution model, deep unification and syntax, associated with implementations of the Prolog programming language. It is implemented in *term* as an encoded token interpreter based on a *solve/unify* loop [3, p. 1313]. The *btrack* to *term* token conversion is performed by "compiling" the *btrack* predicates into *term* rules. Inter-operation with *term* is achieved by defining the predicates that are exported using *term* signatures. The *btrack* compiler then creates the necessary interfaces and entry ports.

Functional Programming Paradigm *Fun*, the functional programming paradigm offers lazy higher order functions supporting currying and call-by-name normal-order evaluation. Its syntax resembles that of Miranda [20] omitting the guard and pattern matching constructs. It is implemented in *bnf*, *lex*, and *term* with function evaluation provided by an *eval/apply* interpreter, written in *term*. Inter-operation with *term* is provided by allowing the import of single result *term* rules and exporting functions as *term* rules with a single result. Calls to and from *fun* need to take into account and respect the *fun* data structuring conventions, which are documented as *term* constructors.

3.3 Experience with *Blueprint*

Blueprint was used to bootstrap itself and, in addition, to implement a small algebraic manipulation system. In the implementation of *blueprint* four of the six available paradigms were used, as illustrated in Table 6. Each row of the table shows a breakdown (in lines of code) of the paradigms used to implement the system named in its first column. The class structure of every paradigm was described in a *paradigm description file* (the table column labelled PDF), which was then processed by a special compiler in a way analogous to the one described in [16] to create the requisite translation tools. Some of the paradigms were implemented using existing tools and their implementation consists of just a paradigm description file.

Table 6. *Blueprint* paradigm implementation summary

Paradigm	PDF	*imper*	*bnf*	*regex*	*term*	*fun*	Total	%
imper	43						43	1.1
term	70	1192	119	84	666		2131	56.0
btrack	60				316		376	9.9
fun	140		305	59	237	43	784	20.6
bnf	95						95	2.5
regex	379						379	10.0
Total	787	1192	424	143	1219	43	3808	100.0
%	18.5	28.0	10.0	3.4	28.6	1.0	100.0	

The algebraic manipulation system deals numerically and graphically with definite integrals and symbolically with indefinite ones. It was implemented at a fraction of time and effort spent to implement an analogous one in Modula-2 by using all six paradigms available under *blueprint*. The breakdown of the system into the paradigms used is illustrated in Table 7.

4 Related Work

In a survey of multiparadigm programming we have identified more than 100 languages and systems that allow programming in more than one paradigm. A number of multiparadigm projects are described in [9], languages based on distributed system architectures are surveyed in [1], and implementations of different paradigms of parallel computer architectures are examined in [19]. Most of the multiparadigm languages attempt to amalgamate the advantages of their constituent paradigms, either in a pragmatic implementation-targeted way, or by developing an underlying theory. Some typical examples are [18, 26, 21]. Our approach differs from those in that we offer an underlying unifying framework for the combination of arbitrary paradigms instead of examining how specific paradigms can be combined. It is a pragmatic approach weak on its theoretical basis. The *compositional approach* described in [30] can also deal with arbitrary paradigms, but is concerned more with the validation of the resulting system. The intimate relation between the target architecture and the programming language that forms the basis of our approach is examined in [27].

Table 7. Algebraic manipulation system implementation

Function	Paradigm	Lines
Symbolic integration	btrack	127
Lexical analysis	regex	47
Expression parsing	bnf	76
Numeric integration	fun	75
Interfacing	term	131
Graph creation	imper	51
Total	blueprint	507

5 Conclusions and Further Work

The abstraction of programming languages and system architectures as object classes provides a unified model for dealing with multiparadigm programming systems. Objects are used as encapsulation entities for modules written in different paradigms, while inheritance is used to bridge the semantic gap between a high level language and the target architecture. Our implementation of an exemplar system based on these principles and its use both as a bootstrapping vehicle and as an implementation platform demonstrated the viability of this approach. We intend to use the same approach on different types of computer architectures in order to test its applicability and limits. One other challenging problem is the development of a theoretical reasoning framework that can be applied to architecture and language systems structured using object classes.

Acknowledgements

We would like to express our thanks to our colleagues at the Distributed Software Engineering group at Imperial College for ideas and discussions during the conduct of this research, and to the anonymous reviewers for their helpful comments. Financial support from the British Science and Engineering Research Council and the DTI under grant ref. IED4/410/36/002 is gratefully acknowledged.

References

1. Bal HE, Steiner JG, Tanenbaum AS (1989) Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21: 261–322.
2. Byrd L (1980) Understanding the control flow of Prolog programs. In *Logic Programming Workshop*, Debrecen.
3. Cohen J (1985) Describing Prolog by its interpretation and compilation. *Commun. ACM*, 28: 1311–1324.
4. Cook WR (1992) Interfaces and specifications for the Smalltalk-80 collection classes. *ACM SIGPLAN Notices*, 27: 1–15. Sevent Annual Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '92 Conference Proceedings, October 18–22, Vancouver, British Columbia, Canada.

5. Debray SK (1988) *The SB-Prolog System, Version 3.0: A User Manual*. University of Arizona, Department of Computer Science, Tucson, AZ 85721, USA.
6. Eckberg DL, Hill , Jr. L (1980) The paradigm concept and sociology: A critical review. In Gutting G, editor, *Paradigms and Revolutions*, pp 117–136. University of Notre Dame Press, Notre Dame, London.
7. Foster I, Taylor S (1990) *Strand: New Concepts in Parallel Programming*. Prentice-Hall.
8. Greene JC (1980) The Kuhnian paradigm and the Darwinian revolution in natural history. In Gutting G, editor, *Paradigms and Revolutions*, pp 297–320. University of Notre Dame Press, Notre Dame, London.
9. Hailpern B (1986) Multiparadigm research: A survey of nine projects. *IEEE Software*, 3: 70–77.
10. Johnson SC (1975) Yacc — yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA.
11. Kuhn TS (1970) *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago and London, 2nd, enlarged edition. International Encyclopedia of Unified Science. 2:(2).
12. Lesk ME (1975) Lex — a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA.
13. Meyer B (1990) Lessons from the design of the Eiffel libraries. *Commun. ACM*, 33: 68–88.
14. Nelson ML (1991) An object-oriented tower of Babel. *OOPS Messenger*, 2: 3–11.
15. Shriver BD (1986) Software paradigms. *IEEE Software*, 3: 2.
16. Spinellis D (1993) Implementing Haskell: Language implementation as a tool building exercise. *Structured Programming*, 14: 37–48.
17. Stefik MJ, Bobrow DG, Kahn KM (1986) Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3: 10–18.
18. Takeuchi I, Okuno H, Ohsato N (1986) A list processing language TAO with multiple programming paradigms. *New Generation Computing*, 4: 401–444.
19. Treleaven PC, editor (1990) *Parallel Computers: Object-Oriented Functional, Logic*. John Wiley & Sons.
20. Turner DA (1985) Miranda — a non-strict functional language with polymorphic types. In Jouannaud JP, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pp 1–16, Nancy, France. Springer-Verlag. Lecture Notes in Computer Science 201.
21. Uustalu T (1992) Combining object-oriented and logic paradigms: A modal logic programming approach. In Madsen OL, editor, *ECCOP '92 Europen Conference on Object-Oriented Programming*, pp 98–113, Utrecht, The Netherlands. Springer-Verlag. Lecture Notes in Computer Science 615.
22. Warren DHD (1983) An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, 333 Ravenswood Ave., Menlo Park, CA, USA.
23. Wegner P (1987) Dimensions of object-based language design. *ACM SIGPLAN Notices*, 22: 168–182. Special Issue: Object-Oriented Programming Systems, Languages and Applications, OOPSLA '87 Conference Proceedings, October 4–8, Orlando, Florida, USA.
24. Wegner P (1989) Guest editor's introduction to special issue of computing surveys. *ACM Comput. Surv.*, 21: 253–258. Special Issue on Programming Language Paradigms.
25. Wegner P (1990) Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1: 7–87.
26. Wells M (1989) Multiparadigmatic programming in Modcap. *Journal of Object-Oriented Programming*, 1: 53–60.
27. Wirth N (1985) From programming language design to computer construction. *Commun. ACM*, 28: 159–164.

28. Wirth N (1985) *Programming in Modula-2*. Springer Verlag, third edition.
29. Wittgenstein L (1960) Philophische Untersuchungen. In *Schriften*, volume I. Suhrkamp Verlag, Frakfurt a.M., Germany. In German.
30. Zave P (1989) A compositional approach to multiparadigm programming. *IEEE Software*, 6: 15–25.