Towards a Standard Family of Languages for Matching Patterns in Source Code

Uri Dekel*
ISRI, School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
udekel@cs.cmu.edu

Tal Cohen
Dept. of Computer Science
Technion – IIT
Haifa, Israel 32000
tal@forum2.org

Sara Porat
Software and Systems
IBM Haifa Research Lab
Haifa, Israel 31905
porat@il.ibm.com

Abstract

This paper makes a case for the definition of a family of languages for expressing patterns over both the structure and semantics of source code. Our proposal is unique in that it attempts to provide a unified solution to the problem of searching for patterns in multiple programming languages, and in that it focuses on the semantics of the program rather than on its syntactic structure, all while striving to ensure simplicity and ease-of-use. We present the motivation and the unique difficulties involved in defining such languages, discuss strategies for dealing with these problems, and propose a prototype family of Code Pattern Languages (CPLs).

1. Introduction

Pattern-matching is a recurring notion in software, with a variety of applications. Standardized pattern languages, coupled with efficient matching tools, allow developers to easily write software that requires pattern-matching capabilities. *Regular Expressions* [12, 23], for example, are the de-facto standard means of matching and transforming strings. They are natively supported by some languages (e.g., Perl [27]), or available as a reusable library in others [14]. Pattern languages are also available for other kinds of data, such as *file wildcards* for file listings, and XPATH and XQUERY for XML data.

In this paper we make a case for the definition of a standardized family of languages for specifying *patterns over source-code*. We describe many potential uses for such languages, and discuss some of the problems unique to defining patterns over code, and in particular patterns that involve multiple PLs. This discussion is followed by design principles for a prototype family of pattern languages, and examples of patterns over the JAVA language.

By "patterns over source-code" we do not refer simply to the syntactic structure of the code, but also to the semantics of the software system that it realizes. Regular expressions and queries over an abstract syntax tree [1] can yield satisfactory results for localized queries over syntax, such as *find all method invocations* or *find a particular macro definition*.[1] Their power is not sufficient, however, for non-localized queries that refer to the semantics of the program. Consider, for example, the following query: *find every direct and indirect subclass of a given class*. To successfully handle such a query, the matching software must construct a model of the class hierarchy, a task clearly beyond the reach of syntax-based solutions. Furthermore, some patterns will be temporal in nature (e.g., *find a particular sequence of method invocations*), which requires the matching software to simulate the flow of control in the program.

Paul and Prakash [26] argue that the lack of adequate models for representing source code information and expressing queries is the primary problem in designing frameworks for source code querying. To this end, they propose a relational framework similar to the use of *relational algebra* [34] in relational databases. We believe, however, that ease of use is not of lesser importance than expressive power, and advocate a declarative textual query language analogous to SQL, which is the de-facto standard for queries over relational databases. Our strategy strives to provide users with a query language which imitates the look-and-feel of specific PLs and also addresses their specific intricacies. Implementation and representation details are encapsulated within the software support for the language.

*Research carried out while author was working for IBM Haifa Research Lab.

[1] The results of localized queries might be inaccurate when syntactic means are used. For example, a simple way to discover method invocations is to search for a name followed by parentheses. This approach might result in irrelevant results, such as function prototype declarations, or even invocations that were commented out.

Many works on querying over code are either limited to a single programming language, or use a specific PL-independent representation of the software, which is manifested in the query language (e.g., [3, 17]). Our goal is ambitious: we wish to design a pattern language that supports a variety of PLs from multiple paradigms, sometimes within the same pattern. The language should address the subtleties of individual PLs while still providing generic constructs for notions that recur across multiple PLs. Furthermore, we want the same infrastructure to be used for implementing pattern matching tools for all these PLs. Clearly, these requirements pose significant challenges, which are described and addressed in this paper.

Our discussion is therefore restricted to the challenges and principles in the definition of the language, and not to strategies for implementing the pattern-matching software that must accompany it.[2] Furthermore, our proposal serves as a blueprint for a proof-of-concept prototype rather than as a complete or optimal solution. We believe that a large scale collaboration is necessary in order to create such standardized language and bring it into general use.

Outline This paper is organized as follows: Section 2 describes the motivation and potential uses of a language for code patterns. The challenges involved in defining such a language are presented in Section 3. In Section 4 we propose some design principles for such a language and describe the CPL family of languages. A prototype sub-language for patterns over JAVA source code, called JCPL, is presented in Section 5. Finally, we conclude and suggest avenues for further research in Section 6.

2. Motivation

This section provides a motivation for the definition of a standardized language of patterns over source code, by describing various applications for such a language and its accompanying tools.

2.1. Enforcement of best-practices

In most programming languages, there are generally-accepted *"best practices"* (e.g., [11, 15, 24]) which programmers should follow to produce quality code. Examples of such practices in JAVA include:[3] *set object references to* `NULL` *when they are no longer needed*; *do not return from a* `try` *block*; *use* `StringBuffer` *rather than* `String` *for concatenation*; etc. Although in many cases such practices can improve the safety and performance, and in some cases even the correctness of the code, only experienced programmers are sufficiently familiar with them and follow them consistently.

For this reason, some compilers and third party software tools (e.g., LINT [7] and PMD [28]) try to discover and report discrepancies from these practices. The rules are often hard-coded into the tool, and the user cannot simply augment them with personal rules or proprietary corporate-standards. Even in tools designed for user-extension, new rules are often limited to specific and inflexible categories (e.g., never invoke a certain method), or must be written as new code that relies on the internal data structures of the tool. For example, the addition of new rules to PMD involves writing new JAVA classes or complex XPATH [36] queries over the internal syntax tree that PMD generates.

In contrast, a robust code pattern language will allow users to easily customize tools such as PMD in a much simpler, standardized manner. All enforcement tools will use the same core language to search for violations of these proprietary rules, and would vary only in their performance and the accuracy of their analysis.

2.2. Enforcement of correct protocol use

A similar problem to the enforcement of best practices is the enforcement of correct simple protocol use. Some third-party libraries and APIs (such as GUI toolkits or database bridges) should be used according to a simple protocol. For example, GUI windows should be created by invoking a certain sequence of operations, database connections should only be accessed after they were successfully opened, etc.

Today, such protocols are often defined only in the documentation. As a result, each checking tool requires protocols to be specified in a proprietary format, and the users of the library are often the ones who have to input the protocol specification in order to check their programs. If there was a standard language and tool for defining and matching program behavior patterns, then API vendors could provide patterns that would describe and specify the correct protocol for using their products. These protocol requirements could then be verified using standard pattern matching tools.

2.3. Software verification

A standardized language for code patterns can also be useful for the enforcement of more complex specifications that require the use of a *model checker* [5]. *Specification languages*, such as LTL and CTL, have long been used in the verification of hardware systems. Model checkers receive specifications that describe properties that must hold throughout the operation of the system, and attempt to discover execution paths that violate them. An example of such a hardware specification is: *signal X must never be active*

[2] Note that many of the potential applications for the described patterns language do not require user interaction. In these cases, the performance of the matching tools is secondary to their ease of use and expressive power.

[3] Items 7,22, and 31 in [11].

during two consecutive cycles. Similarly, one may want to verify a similar property of a software system, such as: *a database connection must not be opened a second time if it is already open.*

One of the difficulties in verifying software with traditional model checkers is in deriving an abstract model of the software system. Even just specifying such a property is difficult because one has to describe the events that will become states. While some extensions (e.g., SUGAR [32]) improve the hardware-specific capabilities of specification languages, we are not aware of similar extensions specific for software that can deal with the intricacies of individual PLs in real-world, large-scale programs.

In the example above, the opening and closure of the database connection are likely to be important individual states of the abstract model, while other events and behaviors of the program will only be a burden. We believe that a pattern language for source code may help specify and pinpoint such events and thus assist in the derivation of an abstract model of the program.

2.4. Software migration and porting

The migration of software across different versions of tools and languages on a certain platform, as well as the porting of software across platforms, typically involves source-to-source transformations.

When this process is carried out manually, a migration expert familiar with both the source and target environments examines the code, looking for constructs that should be changed during the transition. The expert then uses more knowledge about both environments to carry out and test the necessary changes.

Since such a manual porting process is expensive, time consuming, and error-prone, various tools (e.g., [29]) strive to automate it. These tools typically use a knowledge-base of potential porting issues to discover problems and, if possible, to suggest solutions or action paths. When not hard-coded, this knowledge is usually restricted to specific kinds of problems, such as the use of platform-specific API functions and *include files*. Again, it is difficult for a user to add new rules that do not fall into the existing categories.

We argue that the search for many porting problems could be accomplished by a code pattern matcher, which uses a repository of patterns as the porting knowledge base. Furthermore, in a similar manner to substitutions in regular expressions over strings, the patterns can define not only the problems to be sought, but in some cases also the required transformations. The language could be used not only to define replacements for single API calls, but also to carry out transformations between protocols. For example, if opening a window in one GUI toolkit uses a specific sequence of API calls, we can define a transformation to an equivalent sequence in a different toolkit using an appropriate pattern.

2.5. Software refactoring

Refactoring [9, 25] is another kind of source-to-source transformation that helps improve the quality of software. Many tools and development environments [8, 13, 18] provide automatic refactoring services, but these are usually limited to specific hard-coded transformations. A flexible language for specifying code patterns and transformations can be used to enhance such environments, and allow users to customize them with a variety of new transformations.

2.6. Searching software asset repositories

Many organizations use asset management systems [19] to maintain a repository of code assets and reusable components, and many works [30, 38] deal with strategies for searching and retrieving artifacts from such repositories. The retrieval in most commercial tools (e.g., [37]), however, relies on querying for specific properties that are hard-coded into the repository system, such as the names of members, superclasses and implemented interfaces. Some of these tools also allow the user to create more complex searches by combining queries for these properties, but the combination is limited to standard boolean operations over pre-defined fields.

We believe that users often require more flexibility from their retrieval systems. For example, a programmer trying to use a certain library or API may want to search the repository for existing code that uses the same functions from within a similar context, perhaps with specific parameters. In addition, one may want to search for several code artifacts with a certain relation between them, perhaps across language boundaries. Clearly, a simple yet flexible search language will allow users to easily express and carry out such searches.

2.7. Customized Software Metrics

Software metrics [4, 6, 21, 22] are used to quantify the complexity and quality of programs, and to assist in making decisions such as whether it is economical to maintain or migrate a certain software artifact. In some cases, the standard metrics are insufficient, and one needs to measure more specific properties. For example, the frequency of use of macros or templates in C++ source code can hint on the difficulty of porting it to a different compiler or platform. A language for defining code patterns can be used to specify the kinds of properties we are looking for.

3. Challenges

Many challenges complicate the definition of a generic language for code patterns, and the implementation of the software tools that must accompany it. We raise some of these problems here, and will propose strategies for dealing with them in the next section.

Note that in the scope of this paper, we are concerned only with the definition of a declarative language. We do not deal with the many challenges involved in the implementation of supporting tooling such as the pattern-matching software. As we will see, even the seemingly simpler task of defining a language is far from being trivial.

3.1. General vs. language-specific constructs

We argue that defining a language of patterns over source code is a complex task, much harder than defining a language over strings or boolean circuits, for example. Strings may differ in their encoding but are essentially a sequence of values over a set of characters, with no associated semantics. Similarly, hardware-verification languages typically deal with the values of specific signals or groups of signals; a common model language is available.

For source code, our task is more difficult due to the diversity of programming languages. On the one hand, we want to be able to match all the constructs of each specific PL. On the other hand, in order to simplify the pattern language and associated tools, it is preferable to have a generic construct for notions that recur in multiple PLs. Balancing between these two requirements is a significant problem.

Consider, for example, the common notion of inheritance in object oriented programming languages; should the pattern language provide a generic construct for this notion? Inheritance is limited to the OO-paradigm, and is irrelevant when querying over languages from other languages. In addition, even within the limited scope of the OO-paradigm, every language has its own semantics and specific kinds of inheritance, such as non-**`public`** inheritance in C++, or *interface implementation* in JAVA.

Many other notions are difficult to generalize between languages of the same paradigm. Consider the fundamental notion of a *class*. Do JAVA interfaces and C++ **`struct`**s [31] fall under this definition?[4] What about C **`struct`**s [16] and SMALLTALK metaclasses [10]? OO languages also differ in many other aspects, such as the relationships between types and classes (i.e., is every class a type?), method dispatching (e.g., *message* vs. *method* in SMALLTALK), and the existence of *metaclasses*.

[4] A **`struct`** in C++ is equivalent to a class (except for minor encapsulation details), whereas in C it does not even constitute a type definition.

Note that adapting the pattern language to a specific PL involves not only support for particular constructs and semantics, but also an entire grammatical structure that determines the "look-and-feel" of patterns. For instance, in some languages, functions are applied to objects; in others, objects invoke methods. A C++ developer recognizes both `a.f()` and `a->f()` as method invocation syntax, whereas a JAVA developer would not be familiar with the latter variant; a SMALLTALK programmer will only find the construct `a f` familiar.

Clearly, if we want developers to adopt the new pattern language, it should have the look-and-feel of the programming language they are specifying patterns for.

3.2. Spanning multiple languages in a single pattern

The problem raised by the diversity of programming languages is further aggravated by patterns that involve several languages, sometimes from different paradigms. One example of such a pattern would be: *match all the functions in a certain* JSP *file that instantiate a* JAVA *class with at least one method that opens a database connection*. This pattern involves constructs from two languages of different paradigms, as the functions are entities of the JSP scripting language [2], whereas the methods of the class are defined in the object-oriented JAVA language.

The conflicting nature of uni- and multi-language patterns poses a significant design challenge. Our pattern language must be capable of taking the form of specific programming languages in some patterns, while retaining the ability to combine several languages in others.

A single pattern-language which is generic enough to combine constructs from several programming languages in the same pattern will clearly not have the look-and-feel of individual PLs that is necessary for its widespread acceptance. Even if we could create a grammar capable of supporting so many language structures, we would still be faced with the problem of conflicting constructs. The same keywords and lexical elements may have different meanings and grammatical rules in different PLs, making some cross-language patterns impossible to parse. There may even be a conflict between generic keywords in the pattern language (e.g., **`type`**, **`and`**), and the same keywords in specific PLs. Patterns that combine languages with conflicting characteristics, such as those with different type systems or binding times, will be even more difficult to support.

3.3. Extensibility

The challenges presented so far become even more pronounced by the problem of extension. The abundance of existing programming languages, and the emergence of new ones, requires the pattern language and its accompa-

nying tools to be highly extensible in order to apply to as many PLs as possible. Interested parties should be able to adapt the pattern language and tools to handle changes in existing PLs or to add new ones, without disrupting the support for existing languages.

One implication of this requirement, combined with the problems discussed above, is that our pattern language cannot have a complete and fixed grammar. We are even limited in the selection of reserved words and lexical elements due to the possibility of conflict with a future PL.

3.4. Scope and search space

Most pattern languages and search engines are limited in their scopes and search spaces. Regular expressions, for example, are limited to matches over strings; document retrieval systems are limited to the contents of the documents in the repositories and to specific metadata. We argue that the problem of scope becomes a serious concern when pattern matching involves not only the structure of the source code, but also its semantics.

Software interacts with different elements of its environment, such as the system's memory, operating system, persistent storage, and IO devices. This not only complicates the task of the pattern matching software, but also requires more constructs and notions to be brought into the pattern language itself. For example, a pattern such as: *functions that interact with system service* X would necessitate that the notion of a system service be expressible in the pattern language. Other examples include patterns that refer to the contents of files and system pipelines, or to threading primitives that are not part of the programming language per se.

Even when we restrict patterns to a program's structure, the issue of scope must still be addressed. For instance, a C source file is often meaningless without preprocessor definitions, *include files*, and arguments passed to the compiler. Similarly, a JAVA program depends on class paths, environment variables, and parameters passed to the virtual machine. In SMALLTALK, information about the structure of a particular class cannot be fully determined from its source file, because it depends on the exact structure of its superclass at a particular point during runtime.

Hence, the meaning of a pattern may be incomplete without a specification of the *context* in which its target programs operate. A generic language for code patterns must therefore be capable of expressing not only the patterns we are searching for, but also the search-space and scope with which we are concerned. Note that by "context", we do not refer to the actual data against which we are matching, but to other information that is required to fully understand the declarative semantics of the pattern.

4. Design Principles

In the previous section, we presented several problems and concerns that pose significant challenges to the formation of pattern languages for code. We now present our proposed strategy for dealing with these problems by describing CPL, a family of code-pattern languages. Our proposal should be considered as a proof-of-concept prototype rather than as a complete or optimal solution. In particular, no tool-support currently exists for the proposed language. We believe that a large scale collaboration, perhaps in the form of a consortium, is necessary for designing a language which can be standardized and brought into general use, and for developing effective matching tools.

Our discussion interleaves theoretical issues with the actual design of the language and its structure, and is meant to be read incrementally. Each subsection presents an important aspect of the language family. Concrete examples will be presented in the next section for JCPL, a member of the CPL family which is aimed towards defining patterns over JAVA programs.

4.1. CPL is a family of code-pattern languages rather than a single generic language

One of the main questions raised in the previous section concerned the possibility of designing a generic pattern language that still has the "look-and-feel" of specific PLs. As hinted, we believe that this is not possible due to the inherent conflict between PLs. Instead, we suggest a compromise: rather than design a single language, we will create a family of languages. This family will consist of one generic language with maximal expressive power, and many simpler sub-languages specific to particular paradigms or programming languages. From here on we refer to the entire family as CPL.

The primary language in this family, XCPL, will be truly generic: it will allow the specification of patterns for all the supported PLs, including patterns that involve multiple languages; it will also be easily extensible to support new languages. The tradeoff for using XCPL is that it will not be natural to use; the need to support multiple languages makes its grammar very strict and inflexible.

In addition to XCPL, we will have a variety of sub-languages that are limited in their expressive power to a specific domain or to a specific PL. While these languages retain some of the generic concepts of the CPL family, their grammar and structure is closer to the specific PLs they are dealing with. Nevertheless, their expressive power is contained in that of XCPL, and conversion into that language is straightforward. This allows the reuse of the pattern-matching infrastructure provided with XCPL for other sub-languages. One example of such a sub-language, presented

in the next section, is JCPL, a pattern language for JAVA.

4.2. The generic XCPL language is XML based

The need to support multiple programming languages and paradigms (sometimes within the same pattern), as well as the possibility of adding new PLs, discourages us from basing XCPL on traditional PL grammars. Instead, we choose not to tolerate grammatical ambiguities in XCPL patterns, and require users to provide their patterns in the form of a tree. Specifically, all XCPL patterns must be written as XML trees.

XML [35] is a textual markup language that supports the exchange of structured data in the form of trees of markup tags. XML does not specify the available tags, nor their semantics or permitted nesting behaviors. Instead, users define languages or data formats implicitly by the way in which their applications handle the XML data.[5] We chose to use XML instead of proprietary tree formats due to its widespread acceptance, and the availability of parsers and editors.

Since XCPL patterns have no grammar from which a syntax tree must be created, it is easy to add support for new languages and constructs into the grammar. We can define tags which indicate that the elements of a subtree belong to a specific programming language or require the use of some library of special constructs. The software responsible for matching an XCPL pattern will recognize these tags and load the appropriate support when needed.

To see how multi-lingual patterns can be specified in the form of a tree, consider a pattern which involves two languages, such as: *Match all the functions in a certain* JSP *file that instantiate a* JAVA *class with at least one method that opens a database connection*. Figure 1 presents a schematic tree representing the above pattern in XCPL. Characters surrounded by $ signs represent variables and parameters, and are discussed later.

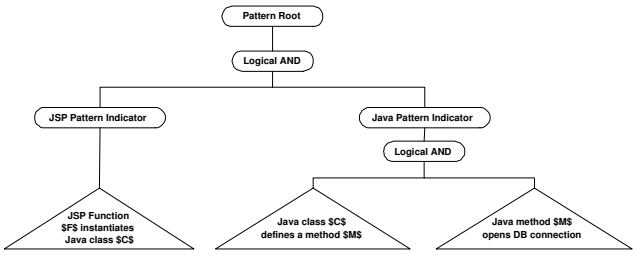

Figure 1. Schematic drawing of an XCPL pattern involving JSP and JAVA constructs

[5] XML also supports the definition of "closed" and well-formed formats using schemas and DTDs. This is not applicable for XCPL which should be open to extensions.

As can be seen in Figure 1, the pattern specification consists of two subtrees, joined by a logical AND. The root of each subtree indicates the specific PL to which the contents of that subtree belong. This structure allows the matching software to interpret the keywords and elements in each tree according to the subtleties of the appropriate PL. In fact, since it is specified that the subtrees contain PL-specific subpatterns, we may not even have to write them as XCPL trees. Instead, we may be able to embed textual specifications in the appropriate sub-languages, such as JCPL. Upon parsing the tree, the matching software will translate the contents of the embedded subpattern into an XCPL subtree using the grammar of the sub-language.

4.3. CPL languages are declarative

Like their counterparts in other domains, all member languages of the CPL family must be declarative. While different matching strategies can be used to deal with complex patterns like the one presented above, matching details are solely the concern of the supporting tools and not of the user. Our vision is for CPL's tooling support to be similar to the programmatic support available for XML and SQL. Developers in various PLs will use a standard set of interfaces or APIs while the actual matching work will be carried out by vendor-supplied solutions.

4.4. Pattern matching provides valuations for variables

A pattern with no variables can be thought of as a predicate; it specifies a certain condition (e.g., whether a certain class inherits from another), and the pattern matches if the condition holds. When we want to obtain a set of values that can match a pattern, such as: *all the* JAVA *methods that open a database connection*, we use variables. When a variable appears once in a pattern, the result of matching that pattern is a (possibly empty) set of valuations rather than a boolean value. In other words, instead of a match confirmation we get a series of matches, each with a different valuation of the variables.

Since matching details will be taken care of by software tools, the simplest way to define a complex pattern is to compose it from several simpler patterns using logical and temporal operators. Variables are then used to connect values in different sub-patterns. If a variable appears in several sub-patterns, it must have the same valuation in all the instances of that variable within each individual match.

Consider, for example, the pattern in Figure 1. This pattern consists of three sub-patterns: **(i)** *The* JSP *function* F *instantiates* JAVA *class* C, **(ii)** *The* JAVA *class* C *defines a method* M, **(iii)** *The* JAVA *method* M *opens a database connection*. The set of valuations for the composite pattern is

a subset of $(F \times C \times M)$.[6] This set will represent all the combinations of functions, classes, and methods that match the original pattern.

Note that a brute force approach to matching these patterns is to simply calculate the valuations of variables in each sub-pattern separately, and then constrain them against each other. As with *joins* in relational databases [34], the efficiency of the matching process can be improved with smarter strategies. For example, we can start by obtaining valuations for M via the third subpattern, use it to obtain the values for C in the second subpattern, and constrain the valuations of C on the first subpattern to obtain values for F. Also, note that not every pattern can be matched with a brute-force approach, since some sub-patterns (e.g., those referring to equivalence between variables) have infinite valuations.

In addition to variables, CPL also supports *parameters*. Whereas variables can be thought of as placeholders for output values, parameters are placeholders for input values. A pattern with parameters is a *template-pattern*; the user supplies valuations for these parameters when attempting to match the pattern. For example, if we often need to search for classes that inherit from other specific classes, we can define a template pattern where the superclass is an input parameter and the subclass is an output variable.

In fact, the success of CPL depends, among other factors, on the availability of a robust library of templates patterns. Reusable templates for common matching tasks can significantly speed up writing complex patterns. A similar successful effort in the field of hardware verification languages is Sugar [32], which does not enhance the expressive powers of CTL and LTL, but simplifies their use by providing new constructs for previously complex specifications.

4.5. CPL constructs have different levels of abstraction

The strategy of creating a specific CPL sub-language for each programming language allows us to address the specific subtleties of PLs. However, to simplify the use of CPL, we want notions that recur in multiple PLs to be expressible using a generic construct shared between the appropriate CPL sub-languages. For example, while JCPL supports patterns over the specific inheritance schemes of JAVA (such as interface implementation), we still want XCPL to provide generic constructs for the common denominator of what constitutes inheritance in object oriented languages, and require support for that XCPL construct from within JCPL. Hence, the same notions may be expressible in several forms and in different levels of abstraction. This implies, of course, that many patterns can be written in a variety of ways.

In our design of the CPL family of languages, we divide constructs into four levels of abstraction, which are presented in Figure 2.

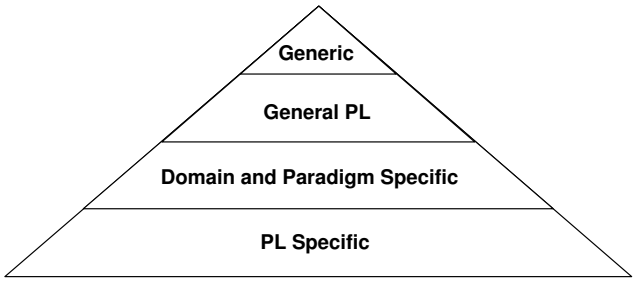

Figure 2. Levels of abstraction of CPL constructs

The *core constructs*, in the top level of Figure 2, are the *meta-elements* of the CPL family. They deal with the way CPL patterns are defined, parsed, and matched, and have nothing to do with the fact that CPL is used to define patterns over source code. Examples of such constructs include elements that serve as the roots of patterns and subpatterns, directives that indicate the use of certain extensions and collections of tags, and elements that define variables and parameters. Generic operators, such as boolean and temporal operators over subpatterns are also found at this layer. Note that from the point of view of the matching software, only core constructs must always be available; support for constructs at lower abstraction levels can be loaded when needed.

Next, we have *general programming languages constructs* which represent fundamental notions available in the majority of PLs, regardless of paradigm. These notions include names, types, variables, constants, control- and dataflow, etc.

In the third level we have the *domain- and paradigm-specific* constructs. They generalize notions that constitute the essence of a particular paradigm. For example, for the procedural paradigm there will be constructs such as procedures, modules, blocks and parameters. The object-oriented paradigm will have constructs such as classes, methods, fields, inheritance, binding, etc. Finally, at the PL-*specific* level we have constructs specific to the unique semantics of the PL (or version therefore) in which we are interested.

The specific sub-languages of the CPL family, such as JCPL, utilize only a fraction of the constructs available in XCPL, with its support for multiple languages. As can be seen in Figure 3, the JCPL sub-language omits not only lower-level constructs, but even some core elements. For instance, pure-JCPL patterns do not need core constructs

[6] Since we are only interested in JSP functions, we will eventually take only the unique valuations of F.

for multi-lingual support.

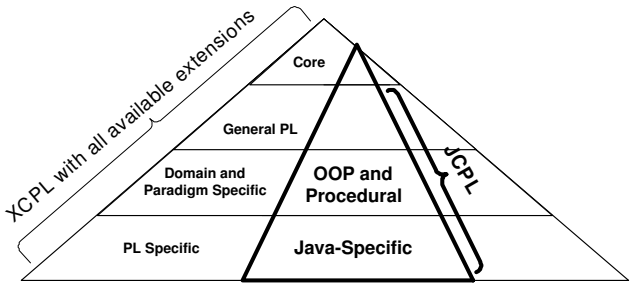

Figure 3. The constructs of JCPL within those of the XCPL language

The constructs available in the lower levels of abstractions are not limited to programming languages. They can also include elements of the environment in which the software operates, such as operating system primitives. In such cases, the third layer may, for example, include notions that recur in most operation systems, such as files and processes, whereas the fourth layer will have representations for the unique primitives or specific OSs. Alternatively, we may think of the hierarchy of non-PL constructs as one that is orthogonal to the hierarchy of the PL constructs; we would then depict the hierarchy as a three-dimensional pyramid.

Note that the borders between the different levels of abstraction are not strict; the same notion can belong to different levels in the pattern languages of different PLs. For example, many design patterns are common to many PLs in the OO paradigm; the third level of abstraction may therefore include constructs for some of these patterns. Nevertheless, some specific PLs may provide native support for these patterns, and these constructs would therefore belong in the fourth level.

5. Examples in the JCPL language

So far we discussed the properties of the CPL family and the structure of the generic XCPL language. While XCPL has the strongest expressive power in its family, most users will use languages that are specific to the PL they are working with. In this section we attempt to demonstrate what patterns over the JAVA programming language may look like. To this end, we rapidly develop a prototype of the JCPL language that can express a variety of interesting patterns in JAVA. The language proposed here does not encompass all the features of JAVA, and may not be fully consistent. We believe, however, that appropriate matching tools for the language proposed here can be constructed using existing components, such as a JAVA parser and an analysis infrastructure which can model the data- and control-flow of JAVA applications (e.g., [33]).

We begin the specification of JCPL with its core constructs. All variables and parameters will have the form of a name preceded by a dollar ($) sign, such as `$aMethod`, `$aClass`, etc. Variables are not explicitly typed; the kinds of element they contain is determined from the context they appear in. Lists of values are specified using square brackets (`[]`) and assigned to variables. For example, `$var[0]` indicates the first item in the list denoted by `$var`.

The common boolean operators, **`and`**, **`or`**, and **`not`**, are available in JCPL; so are the quantifiers over variables, **`forall`** and **`exist`**. Finally, a pattern specification takes the following form:

```
pattern patternName ($output-vars) {}
```

Template patterns take a slightly different form:

```
template ($input-parameters)
  pattern patternName ($output-vars) {}
```

Variables not specified as output variables or input parameters are considered as local variables.

Elements in JCPL have properties, predicates, and functions that can be accessed using the familiar syntax for member access. For example, all elements have the `name` property, so `$var.name` is the name of the entity represented by `$var`. Every method element has attributes named `retval`, `numargs` and `args`. We can also add new properties to existing element types using patterns. Consider for example, elements that represent method invocation instructions in JAVA bytecode [20]. If we want to access the arguments passed to a method during its invocation, we can add an `.args` property to these elements.[7]

We now turn to defining the predicates of the language. Most predicates are either unary or binary, and operate on specific types. For example, the pattern:

```
$aClass isSubclassOf $anotherClass
```

indicates that the class `$aClass` is a direct subclass of `$anotherClass`.

Some predicates can receive qualifiers. We adopt the convention of regular expressions, so that **`isSubclassOf+`** indicates that the first class is either a direct or an indirect subclass of the second, and **`isSubclassOf*`** also allows the first class to be identical to the second.

The JCPL language will provide many built-in predicates; several of them are listed in Table 5.

In addition to the built-in predicates, users can define new predicates in the form of template patterns that have no output variables (and hence return a boolean value). For example, let us define a new predicate named `overridesSpecific`, which indicates whether a particular method overrides a version from a specific superclass:

[7] This property is realized using a pattern that matches the last objects pushed into the stack by the preceding bytecode instructions.

Category	LHS	Predicate	RHS	Qualifiers	Semantics	Qualifier Semantics
General	any	is	any		Equivalence	
	any	named	any		Name equivalence	
Declarative	method	returnsType	type	+	Declared return type is *type*	Return type may be a supertype
	method / field	declaredBy	class	+	Method or field declared in class	Declaration in superclasses
	class	inPackage	package	+	Class belongs in package	Class can be in enclosed packages
	field / method	isPublic / isProtected			Method or field has public or protected encapsulation	
	field / method	isPrivate / isPackage			Method or field has private or package encapsulation	
	field / method	isStatic / isFinal			Method or field is static or final	
	class / method	isAbstract			Class or method is abstract	
	class	isInterface			Class is an interface	
	method	isConstructor / isFinalizer			Method is constructor or finalizer	
Typing	class	implements	interface	+	Class implements interface	May implement derived interface
	class	isSubclassOf	class	+ / *	Class is a subclass of another	Subclasses indirectly / Is identical
	method	overrides	method	+	Method directly overrides another method	May override indirectly
	variable	isOfType	type		Variable has specific type	
	object	instanceOf	type	+	Object is an instance of class	May be instance of subclass
Control Flow	JVM instruction	executesBefore / executesAfter	JVM instruction	+	JVM instruction executes before / after another	Nonconsecutive instructions
	JVM instruction	executesWithin	method	+	Method contains or executes instruction	Covers invoked methods
	method	invokes	method	+	Method invokes another method	Covers indirect invocation
Data flow	instruction / method	changesValueOf	variable / field		JVM instruction / method writes value	
	instruction / method	returnsValue	variable / field		JVM instruction / method has value	

Table 1. Various predicates in the JCPL language

```
template(m,c)
  pattern overridesSpecific() {
    $m overrides+ $n and
    $n declaredBy $c
}
```

Let us put more predicates from Table 5 to use, and write a useful real-life pattern. Many "best-practices" take the form of *do not invoke method X from Y*. One such practice is: *do not call any method of class `java.lang.Runnable` from within `finalize()`*. A pattern that matches violations of this practice is:

```
pattern RunnableFromFinalize () {
  $x invokes+ $y and
  $x isFinalizer and
  $y declaredBy "java.lang.Runnable"
}
```

Another best-practice is: *do not pass a non-serializable object as the second argument to `HttpSession.setAttribute()`*. Following is a pattern that finds all the methods which violate this practice:

```
pattern NonSerializableInHttpSession(m)
{
  $inv executesWithin $m and
  $inv.kind in [ invokevirtual,
    invokeinterface, invokespecial ]
  and $inv.args[1] is $o
  and not ($o implements+
      "java.lang.Serializable")
  and $m declaredBy
    "javax.servlet.HttpSession"
  and $m.descriptor is
    "setAttribute(String, Object):void"
}
```

The above examples give concrete motivation for a pattern language over JAVA, and conclude our discussion of JCPL in the scope of this paper. We believe that the prototype developed in this section can serve as a reference model for the implementation of a pattern language and supporting software for JAVA.

6. Conclusions

As we pointed out in the introduction, the goal of this paper is not to provide a complete and consistent solution, but to present the motivation and potential benefits of one. We showed how a generic language with the capabilities we are proposing can assist in a variety of tasks. To the best of our knowledge, no solution of this scale exists or is being developed at this time. Developers who require the capabilities we suggested rely on partial or ad-hoc solutions, such as using syntax-based tools. We believe that code pattern languages, coupled with sufficient tool support, can become highly reusable components in many applications. Our hope is to see CPL processing components available in the future in the same way that out-of-the-box XML parsers are available today.

The economic benefits of such a standardized infrastructure for code patterns is clear, assuming that the task of developing it is feasible. In our discussion we presented the main challenges that face us, addressed them in depth, and suggested compromises when clear conflicts arose. It is our hypothesis that our proposal is sound, and can serve as a basis for an actual and successful language.

Naturally, this hypothesis cannot be verified without working out the details of the languages and implement-

ing supporting tools; this is clearly a matter of further research that requires more time and resources before a working prototype is available. Furthermore, even if the proposed language is feasible, the performance of the accompanying matching tools is a serious factor in its chances of general acceptance. Specification languages for hardware were not useful before model checkers were scalable to the systems they were supposed to verify. Similarly, much research is required to discover efficient matching strategies that are appropriate to the unique problem of searching patterns in multiple programming languages.

Acknowledgements

The authors would like to thank Yuri Tsoglin for his role in the formation and definition of the JCPL language.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[2] H. Bergsten. *JavaServer Pages*. O'reilly, 2nd edition, 2002.

[3] R. Bull, A. Trevors, A. Malton, and M. Godfrey. Semantic grep: Regular expressions and relational abstraction. In *Proceedings of the 9th working conference on reverse engineering*, pages 267–276. IEEE Computer Society Press, Oct. 2002.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[6] T. Cohen and J. Y. Gil. Self-calibration of metrics of Java methods. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, pages 94–106, Sydney, Australia, Nov. 20-23 2000. TOOLS Pacific 2000, Prentice-Hall.

[7] I. F. Darwin. *Effective C++*. O'reilly, 1st edition, 1998.

[8] Eclipse project homepage. `http://www.eclipse.org`.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[10] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[11] P. Haggar. *Practical Java*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[12] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[13] IntelliJ IDEA homepage. http://www.intellij.com/idea/.

[14] JAKARTA-ORO homepage. `http://jakarta.apache.org/oro/`.

[15] JAVA practices collection website. `http://www.javapractices.com/index.cjp`.

[16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, 2nd edition, 1988.

[17] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *Proceedings of the 5th working conference on reverse engineering*, pages 135–143. IEEE Computer Society Press, Oct. 1998.

[18] R. Lämmel. Towards generic refactoring. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, pages 15–28. ACM Press, 2002.

[19] W. C. Lim. *Managing Software Reuse*. Prentice-Hall, 1998.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1999.

[21] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[22] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, Dec. 1976.

[23] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE trans*, EC-9:39–47, March 1960.

[24] S. Meyers. *Effective C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1997.

[25] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[26] S. Paul and A. Prakash. Querying source code using an algebraic query language. In H. A. Müller and M. Georges, editors, *Proceedings of the IEEE International Conference on Software Maintenance*, pages 127–136, Victoria, B.C., Sept. 1994. IEEE Computer Society Press.

[27] PERL homepage. `http://www.perl.com`.

[28] PMD project homepage. `http://pmd.sourceforge.net`.

[29] Porting Manager homepage at IBM alphaworks. http://www.alphaworks.ibm.com/tech/portingmanager.

[30] Y. Shen and Y. Park. Concept-based retrieval of classes using access behavior of methods. In *Proc. of the Int. Conf. on Inf. Reuse and Integration*, pages 109–114, 1999.

[31] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, Mar. 1994.

[32] SUGAR homepage. `http://www.haifa.il.ibm.com/projects/verification/sugar/index.html`.

[33] TOAD homepage at IBM alphaworks. http://www.alphaworks.ibm.com/tech/toad.

[34] J. D. Ullman. *Principles of database and knowledge-base systems*, volume 1. Computer Science Press, 1988.

[35] XML homepage. `http://www.w3.org/XML/`.

[36] XPATH homepage. `http://www.w3.org/TR/xpath`.

[37] A. Yaeli, A. Akilov, I. Ragoler, S. Porat, S. Shachor-Ifergan, and G. Zodik. Asset Locator - a framework for enterprise software asset management. In *The Israeli Workshop on Programming Languages and Development Environments*, Haifa, Israel, July 2002. IBM Haifa Research Lab. `http://www.haifa.il.ibm.com/info/ple/`.

[38] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, 1997.