

# Closing the Gap between Modelling and Java

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende

Technische Universität Dresden  
Institut für Software- und Multimediatechnik  
D-01062, Dresden, Germany

`florian.heidenreich|jendrik.johannes|mirko.seifert|c.wende@tu-dresden.de`

**Abstract.** Model-Driven Software Development is based on standardised models that are refined, transformed and eventually translated into executable code using code generators. However, creating plain text from well-structured models creates a gap that implies several drawbacks: Developers cannot continue to use their model-based tool machinery, relations between model elements and code fragments are hard to track and there is no easy way to rebuild models from their respective code.

This paper presents an approach to bridge this gap for the Java programming language. It defines a full metamodel and text syntax specification for Java, from which a parser and a printer are generated. Through this, Java code can be handled like any other model. The implementation is validated with large test sets, example applications are shown, and future directions of research are discussed.

## 1 Introduction

The goal of Model-Driven Software Development (MDSD) is the (semi-)automatic generation of software systems from models across multiple stages [1, 2]. That is, not only code—such as Java source code—is generated from models, but also models are transformed and refined towards other models. Using a standardised metalanguage, language-independent tools can be utilised to manipulate and analyse models defined in different modelling languages. Since a metamodel defines types and constraints for sentences (i.e., models) of a language, all model manipulations done by different tools can be checked for correctness.

As described above, almost all transformations in an MDSD process produce structured and typed data, even on the level of abstract system modelling (e.g., Use Case modelling). However, the last transformation, from models to code artefacts, is often done in a weak structured and untyped manner using string processing template engines. This is a paradox since type checking and correctness is most important when producing compilable artefacts.

In addition, many modellers—in particular the ones involved in the last steps of an MDSD process—are also programmers. Today’s common practices, such as annotating models with (again untyped) Java code, show that a tighter integration between modelling and programming languages is often desired.

We argue that there is a *gap* between modelling and programming languages that is worth closing to tackle many of today’s problems in the last steps of

MDSO processes. The *gap* is caused by the fact that modelling and programming languages are too often regarded as different things. If a programming language like Java would be handled equally as other modelling languages, the issues discussed above could be addressed: existing modelling tools could handle Java programs as they handle other models—structured and typed—instead of treating them as plain text. By using metamodelling tools for extension and reuse of language specifications, Java (or parts of Java) can be integrated with other modelling languages. As a consequence Java can be utilised as any other modelling language.

To *close the gap* for the Java programming language, we propose the Java Model Parser and Printer (JaMoPP). JaMoPP leverages Java to a modelling language by providing the following:

1. JaMoPP defines a complete metamodel for Java that covers the whole language. The metamodel is defined in the commonly used metamodelling language Ecore [3] which allows it to be processed by metamodelling tools for custom modification, extension or reuse.
2. JaMoPP defines a text syntax that conforms to the Java language specification and from which a parser—to create instances of the metamodel from Java source code—and a printer—to transform instances of the metamodel into Java source code—are generated. Similar to the metamodel, the text syntax—being a model on its own—can be customised, extended and reused and the tooling (i.e., parser and printer) can be regenerated.
3. JaMoPP’s Java metamodel reflects the static semantics of Java through cross-references between model elements. These references are established after parsing by an analysis mechanism that implements the specifics of Java’s static semantics. This mechanism is implemented in a modular fashion to support customisation, extension and reuse.

With JaMoPP, Ecore-based modelling tools can process Java files in the same manner they process other models. Additionally, the same tools can be applied to the Java metamodel itself. We explore different scenarios later in this paper.

The paper is structured as follows: Sect. 2 gives details about the design and implementation of JaMoPP. This includes our metamodel for the Java language, the text syntax specification, the static semantics analysis, the integration of compiled Java classes and details about the extensive test suite that was used to validate our implementation. Section 3 discusses different *closing the gap* problems that can be tackled by JaMoPP. We compare our work to existing approaches in Sect. 4 and draw conclusions in Sect. 5.

## 2 Overview of JaMoPP

This section introduces the different parts of JaMoPP. An in-depth description of JaMoPP can be found in [4]. In Sect. 2.1 we discuss our Ecore metamodel for Java. Section 2.2 presents details of an EMFText [5] syntax specification for Java, the tooling generated based on this specification and the static semantics

analysis implementation. Finally, Sect. 2.3 gives details about the process used to test the different parts of JaMoPP.

## 2.1 Java Metamodel

There is a huge amount of tools that operate on Java programs and therefore implicitly or explicitly operate on instances of the Java metamodel. Despite the great variety of software that depends on the Java metamodel, it turned out few have an explicit representation of it.

The Java Language Specification (JLS) [6] itself does not provide a formal metamodel of Java. Existing Java parsers (e.g., `javac` or the Eclipse Java Development Tools (JDT)) have internal metamodels written in Java. One implementation that is closest to a standardized solution is the Java 5 implementation of the Stratego/XT system [7]. However, none of these implementations does provide an integration with standard metamodeling tools.

In contrast, the Java metamodel published by the OMG [8], the MoDisco<sup>1</sup> project, or the SPOON [9] project are based on a standardized metamodeling language (in particular Ecore), but did not fulfill our need for completeness. Thus, we decided to compare the existing metamodels, extract commonalities and extend them to fully support the JLS. The complete metamodel is available online at the JaMoPP Website<sup>2</sup>.

Our metamodel defines 80 abstract and 153 concrete classes, which are divided into 18 packages. Our metamodel contains all elements of the Java language (e.g., classifiers, imports, types, modifiers, members, statements, variables, expressions and literals) including those that were introduced with the release of Java 5 (e.g., annotations and generics).

## 2.2 Java Syntax and Static Semantics

To put our metamodel into practical use, a text syntax specification is needed from which tooling can be generated. This generation step takes text syntax rules defined in EMFText's specification language CS as input, which were defined for each concrete metaclass. Showing the rules here is not possible due to space limitations, but the complete set of rules is available on the JaMoPP Website. The derivation and implementation of the tool components (i.e., a parser, a printer, and a set of reference resolvers) are explained next.

**Parsing** is based on a Context-free Grammar (CFG), which is derived from the CS specification and implemented using a recursive descending parsing strategy as explained in [5]. Although Java is not completely context-free, the backtracking mechanism of the utilised parser generator ANTLR<sup>3</sup>, which is a back-end for EMFText, allowed us to specify the complete Java grammar according to the specification and generate the parser from it.

---

<sup>1</sup> <http://www.eclipse.org/gmt/modisco>

<sup>2</sup> <http://jamopp.inf.tu-dresden.de>

<sup>3</sup> <http://www.antlr.org>

**Reference Resolving** corresponds to static semantics analysis in Java models. EMFText generates *reference resolvers* for all non-containment references, which replace symbolic names in the tree-structured parse model by explicit links to the corresponding elements. Non-containment references extend the context free structure of the parse tree to represent the context sensitive results of static semantics analysis. Reference resolvers can be manually refined if a language has specific scoping rules or requires the loading of additional resources. Thus, we refined the references resolvers with respect to Java’s static semantics.

Because of the high fragmentation of Java models into several files there are many non-containment references. JaMoPP uses a global registry, which corresponds to a Java classpath, to manage resources and their physical location. This registry is used to find cross-referenced model resources on demand and, thus, enables reference resolving across multiple files. In addition, references may point to classes in a library that is only available in byte code format. For reference resolving in this setting, we use the BCEL<sup>4</sup> byte code parser and translate the output of the BCEL parser into an instance of the JaMoPP metamodel.

**Printing Java Source Files** is the inverse process to parsing. EMFText generates a printer from the CS specification that contains one separate print method for each concrete metaclass. According to the CS rule that belongs to a class, the printer emits keywords for model elements, the values of element attributes and recursively calls subsequent methods to print contained elements.

**Tool Integration** Ecore-based modelling languages and tools are integrated into the Eclipse platform by the Eclipse Modelling Framework (EMF) [3]. New languages can be transparently integrated into this infrastructure by implementing EMF’s **Resource** interface. Therefore, JaMoPP provides a **JavaResource** for \*.java and \*.class files that makes use of the generated parser and printer, the reference resolvers and the byte code parser to load and store Java models. Thus, despite of their specific text syntax, Java models can be handled as any other models by EMF-based tools.

### 2.3 Test and Evaluation

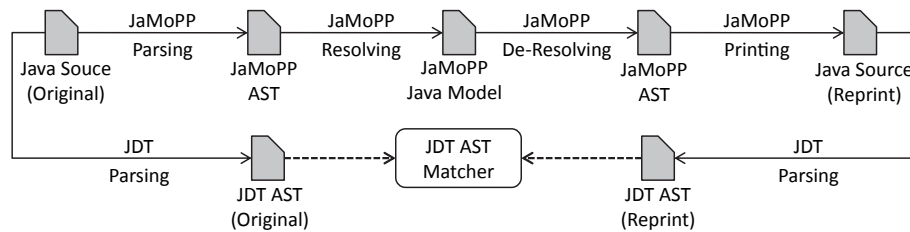
The previous sections presented our metamodel, as well as a parser and a printer for the Java language. To show that our approach can handle industrial-sized applications, a large test suite and a test process were created that allow to check whether JaMoPP complies to a reference implementation (the JDT).

The objectives of our test suite are to verify that 1) our parser accepts valid Java programs, 2) our resolver resolves all names and types, 3) the so created model instance has the expected structure, 4) our resolver de-resolves all cross-references to their correct string representation and 5) our printer emits correct and complete source code for model instances. To meet these five objectives the test process shown in Fig. 1 was employed.

Starting with a valid Java source file (upper left corner of Fig. 1), both our parser and the reference implementation (the JDT) process the given input file

---

<sup>4</sup> <http://jakarta.apache.org/bcel>



**Fig. 1.** Test process for validating printer and parser

and create an Abstract Syntax Tree (AST)—which in the case of our parser is a model with unresolved cross-references. Next, our resolver first resolves all names to cross-references and afterwards de-resolves the cross-references to names again. Then the model is printed to its text form, which is again processed by the JDT parser (lower right corner of Fig. 1). The second JDT AST is then compared to the original JDT AST using the JDT’s own AST matcher.

The given procedure meets the five objectives given above. 1) If our parser does not accept a valid file it can either throw a parsing exception or run forever—both cases are detected by unit tests. 2) After resolving, the test checks the model for unresolved references. With this, failed and wrong resolvings are detected since a wrong resolve often leads to follow-up failures when resolving other elements that depend on the previous resolving. 3) The model instance is checked for completeness through several mechanisms. Elements that are referenced but missing are already detected by the resolver. Furthermore, since the resolving is a complex procedure, other structural errors in the model lead to failures in the resolver. Any other missing information can not be printed and will thus be detected by the AST matcher. In addition, we manually wrote unit tests for distinct Java language features. They consist of assertions that check whether the correct model elements are created for given pieces of source code. 4) When resolving succeeded, the references are de-resolved again and printed. 5) Other errors that are caused by wrong printing only are also detected by the AST matcher. When the matcher delivers a failure, we manually compared the original and the reprint to discover the location of the error.

The input for this test process were 79.017 Java files (15.5 million non-empty lines including comments). In the end, all 79.017 Java files passed the test process. Among these files were some self-defined classes for testing individual language features, the sources of two IDEs (Eclipse 3.4.1 and Netbeans 6.5.1), application and web servers (Apache Tomcat 6.0.18 and JBoss 5.0.0 GA), math libraries (Apache Commons Math 1.2, Mantissa 7.2), web frameworks (Google Web Toolkit 1.5.3, Spring 3.0.0M1 and Apache Struts 2.1.6), an XML Parser (XercesJ 2.9.1), a code generator framework (AndroMDA 3.3), the Sun JDK 1.5.0 Update 16, as well as subsets<sup>5</sup> of the compiler test suite JACKS and the JDT test project. Links to all test source can be found on the JaMoPP Website.

<sup>5</sup> Only compilable classes were used, invalid files were omitted

The tests were automated using JUnit and revealed many errors during the specification of the text syntax and the implementation of the resolvers. Due to the specific nature of the test inputs, quite different classes of errors were found. For example, the tests in the JACKS suite contain many special corner cases mentioned in the JLS. Escaped unicode sequences used in keywords and complex number literals are just a few examples.

In summary we can say that the presented test procedure does not guarantee the correctness or completeness of our implementation, because no proof was established. Nonetheless, the executed tests were performed on industrial-sized applications and give confidence that JaMoPP can be applied in practice.

### 3 Closing the Gap between Modelling and Java

JaMoPP is about *closing the gap* between the programming language Java and modelling languages. The problem we tackled in this paper so far was that Java programs were until now not represented in the same format as models—because no Ecore-based metamodel for the complete language and no parser and printer, to handle the conversion between text and instances of this metamodel, existed. Now that we have these components available, we can look more closely at the research challenges that emerge by *closing the gap*.

We have developed a set of applications that apply JaMoPP in combination with different modelling methodologies and technologies. In the following, we discuss the applications most relevant from a research perspective. For a whole list of applications, descriptions and running examples, please refer to [4] and JaMoPP’s Applications Website.<sup>6</sup>

This section consists of two parts. Section 3.1 discusses advantages of having a complete Ecore-based metamodel and a complete text syntax specification for Java, where both can be extended, modified or (partially) reused through metamodelling. Section 3.2 discusses implications of working with Java programs and models in a unified fashion with existing language-agnostic modelling tools.

#### 3.1 Benefits from metamodelling the Java language

Having Java defined as an Ecore metamodel together with a syntax specification (which is a model itself) allows us to integrate the Java language with other modelling languages directly using metamodelling. In general, there are two directions that can be followed: 1) language extension (as discussed in [10]) where new (domain-specific) constructs are added to the Java metamodel or 2) language reuse where the Java metamodel and syntax are embedded into other (domain-specific) languages. In the following, we discuss one example for each case, both tackling *closing the gap* problems.

**Extending Java for safe code generation** A prominent issue when generating code using model-2-text (m2t) transformations, is that those transformations generate not necessarily correctly structured text and are not aware of the

---

<sup>6</sup> <http://jamopp.inf.tu-dresden.de/applications>

metamodel of their output. In contrast, model-2-model (m2m) transformations transform models (i.e., instances of metamodels) into other models and therefore ensure correctly structured results. With JaMoPP, 1) Java code can be generated by m2m transformations—using for example an ATL<sup>7</sup> transformation.<sup>6</sup> or 2) By building a metamodel-aware template engine as shown in [11].

**Reusing Java statements for behaviour modelling** This JaMoPP application concerns behaviour modelling for operations in UML. While UML includes different behaviour modelling paradigms, it lacks a concrete syntax for specifying actions and behavior. For certain cases, however, it is most convenient to use this imperative programming concept to define behaviour. Today this is often done by attaching a fragment of Java or C code to the UML Operation as plain text that is later injected into the generated code. Here a similar problem as above occurs: The Java code is not recognised as such and errors are only identified in the generated code. Reusing parts of JaMoPP, UML can easily be extended with a Java-based action language.<sup>6</sup>

### 3.2 Benefits from modelling Java programs

Treating Java as a modelling language also allows for reusing modelling tools with Java. We applied JaMoPP in combination with various modelling tools. Our investigations revealed that using these tools with Java programs is benefiting in multiple directions: 1) Java programs treated as models benefit from tools that are only available at modelling level. 2) Java models constructed by modelling tools can be mapped to their textual representation. Thus, tools that only produce models can easily be extended to work with Java. 3) Both directions can be used tightly integrated to enable full round-trip for Java programs and models. One example of each direction is given in this section.

**Generic Source Code Analysis** Using model-based representations of programs, source code analysis can be performed uniformly for arbitrary languages. For example, our analysis tool RestrictED [12] uses declarative expressions written in the Object Constraint Language (OCL) to specify undesired code patterns. In previous work [12] we showed that static analysis can be performed on small modelling languages. With the advent of JaMoPP we applied this approach to the Java language in its full extent.<sup>6</sup>

**Integrated Mapping of Features to Artefacts** In Software Product Line Engineering (SPLE) [13], the FeatureMapper<sup>8</sup> allows for mapping features of feature models to artefacts of EMF-based modelling languages [14]. Since the tool was initially built to support modelling languages only, textual programming languages were not supported. JaMoPP enabled the application of the FeatureMapper to Java programs and enhanced its granularity from modelling in the large to tiny elements in source code. Thus, features can be mapped to both models and source code in an integrated way.<sup>6</sup> To actually derive a specific

---

<sup>7</sup> <http://www.eclipse.org/m2m/at1/>

<sup>8</sup> <http://www.featuremapper.org>

variant of a Java program a mapping of the Java models that are generated by the FeatureMapper to their textual representation is crucial.

**Graphical Editors for Java** Visual representations raise the level of abstraction and help the comprehension of software systems [15]. Thus, especially for modelling languages graphical syntax enjoys a high reputation. To ease the time-consuming and cumbersome task of building visualisations or even graphical editors, several model-based visualisation frameworks emerged [16, 17]. With JaMoPP these technologies can now be applied to build tailored visualisations for Java programs. For instance, the capabilities of the editors generated with the Graphical Modeling Framework (GMF) go beyond just visualising models, but also allow for editing models using their graphical representation. Since the JaMoPP printer serialises Java models to source files, these changes are transparently mapped to the underlying source code.<sup>6</sup> This shows the potential of connecting Java models and their textual representation in both directions.

## 4 Related Work

Closing the gap between the model of a system, specified using a modelling language, and its implementation, written in a programming language, needs to consider two directions: First, getting from the model to the code and, second, getting from the code to the model. In the following we will relate the JaMoPP approach to tools typically used in one or both directions.

Computer-Aided Software Engineering (CASE) tools typically utilise template languages that emit string literals to generate Java code from models. It is hard to ensure that the code generated from these templates is a valid program, since the templates do not regard the syntactic structure of Java. JaMoPP follows the suggestion of [18] and enables the use of an object-language aware template language. This ensures a well-formed structure of generated code, enables tracing of links between the models and the code, and includes the way back from the generated code to the models, which otherwise typically requires additional tools for reverse engineering.

Existing Java compilers, primarily `javac`, `Jikes`, and `GJC`<sup>9</sup>, use internal “models” to represent Java code, i.e., they represent them by a set of Java classes. Thus, they create their own technical space. In contrast, the standardised, Ecore-based representation of abstract syntax in JaMoPP, allows to coordinate different activities and reuse Ecore tools for software development with Java. The most complete, Ecore-based metamodel we found for Java was specified in the SPOON project. SPOON relies on the Eclipse JDT to parse source code and print modifications. This impedes with our goal of extending Java.

From the number of parser and printer generators that exist to provide textual syntax for models EMFText was chosen for implementing JaMoPP—in addition to the fact that we are familiar with the tooling—because of: a) its resolving mechanism that can be tailored to the needs of a complex language as

---

<sup>9</sup> <http://java.sun.com>, <http://jikes.sourceforge.net/>, and <http://gcc.gnu.org/java/>

Java b) its capability to generate printer and parser from a single specification  
c) its import mechanism, which allows for modularisation, extension and (partial) reuse of the syntax specification d) its tight integration with EMF through which developers using EMF-based tools can directly profit from JaMoPP. To the best of our knowledge, none of the other tools analysed in [19] provides all this functionality in combination.

## 5 Conclusion

Recent publications (e.g., [20]) show that treating a programming language as modelling language is needed to close the gap that was introduced by traditional code generation techniques. The JaMoPP approach contributes 1) a comprehensive metamodel of Java defined in the widely used metalanguage Ecore that captures the static semantics of Java in cross-references, 2) an extensible text syntax specification from which parser and printer are directly generated and 3) a modular implementation of Java's static semantics analysis. All components conform to the JLS and cover the complete Java language from high-level constructs like `Package`, `CompilationUnit`, or `Class` to the lowest granularity of individual `Operators` including also comments. Therefore, JaMoPP exceeds the level of detail of other approaches for connecting specification models with (Java-based) system implementations. JaMoPP was tested for completeness with an extensive test suite. This shows that the idea of representing source code as models is feasible and can therefore be transferred to other programming languages.

With JaMoPP, we closed the gap between modelling languages and the implementation language Java. We showed how different problems caused by this gap are tackled with JaMoPP. On the meta-level, the JaMoPP specifications can be extended or reused to enhance the quality of code-generation templates or to reuse parts of Java in other modelling languages. The model-based representation of Java code enables the usage of MDS tools in areas like software visualisation or software product line engineering.

The presented applications can only indicate the full potential of JaMoPP. They encourage further investigations and open new perspectives to leverage MDS. To name a few, metamodelling, (bi-directional) model transformations, model-based tracing and other MDS concepts can now be used to build type-safe Java template languages, separate hand-written from generated Java code or to improve round-trip engineering of Java programs. Generally spoken, Java can now be treated as a modelling language and therefore integrated into MDS as deeply as any other modelling language.

## Acknowledgement

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081, the FP7 project MOST #216691 and by the German Ministry of Education and Research (BMBF) within the projects feasiPLe and SuReal.

## References

1. Ritsko, J.J., Seidman, D.I.: Preface. IBM Systems Journal – Special Issue on Model-Driven Software Development **45**(3) (2006)
2. Völter, M., Stahl, T.: Model-Driven Software Development. Wiley & Sons (2006)
3. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2008)
4. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: JaMoPP: The Java Model Parser and Printer. Technical Report TUD-FI09-10 August 2009, Technische Universität Dresden (2009)
5. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Proc. of the 5th Europ. Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009). Volume 5562 of LNCS., Springer (2009) 114–129
6. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The (3rd Edition) (Java Series). Addison-Wesley Professional (2005)
7. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72**(1-2) (June 2008) 52–70 Special issue on experimental software and toolkits.
8. Object Management Group: Metamodel and UML Profile for Java and EJB Specification Version 1.0. formal/2004-02-02 (2004)
9. Pawlak, R.: Spoon: Compile-time Annotation Processing for Middleware. IEEE Distributed Systems Online **7**(11) (2006)
10. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR) **37**(4) (2005) 316–344
11. Heidenreich, F., Johannes, J., Seifert, M., Wende, C., Böhme, M.: Generating Safe Template Languages. In: Proc. of the 8th Int'l Conf. on Generative Programming and Component Engineering (GPCE 2009), ACM (2009)
12. Seifert, M., Samlaus, R.: Static Source Code Analysis using OCL. In Cabot, J., Van Gorp, P., eds.: Proc. of the MoDELS 2008 Workshop on OCL Tools: From Implementation to Evaluation and Comparison (OCL 2008). (2008)
13. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer (2005)
14. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: Comp. Proc. of the 30th Int'l Conf. on Software Engineering (ICSE 2008), New York, NY, USA, ACM (2008) 943–944
15. Diehl, S.: Software Visualization. Springer (2007)
16. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Towards Graph Transformation based Generation of Visual Editors using Eclipse. Proc. of the Workshop: Visual Languages and Formal Methods (VLFM) (2004)
17. GMF Team: Graphical Modeling Framework. <http://www.eclipse.org/gmf/>
18. Bravenboer, M.: Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates. PhD thesis, Utrecht University, Utrecht, The Netherlands (2008)
19. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Proc. of the 4th Europ. Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008). Volume 5095 of LNCS., Springer (2008)
20. Angyal, L., Lengyel, L., Charaf, H.: A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. In: 15th IEEE Int'l Conf. on Engineering of Computer Based Systems (ECBS 2008), IEEE Computer Society (2008) 463–472