

Generative Programming

Krzysztof Czarnecki¹, Kasper Østerbye², and Markus Völter³

¹ DaimlerChrysler Research and Technology, Germany
czarnecki@acm.org

² IT University, Denmark
kasper@it-c.dk

³ independent consultant, Germany
voelter@acm.org

Abstract. This report describes the results of a one-day workshop on Generative Programming (GP) at ECOOP'02. The goal of the workshop was to discuss the state-of-the-art of generative techniques, share experience, consolidate successful techniques, and identify open issues for future work. This report gives a summary of the workshop contributions, debates, and the identified future directions.

1 Introduction and Overview

The Workshop on Generative Programming was held on June 10, 2002, at the ECOOP'02 conference in Málaga, and it was the second ECOOP workshop on this topic. The workshop aimed to bring together practitioners, researchers, academics, and students to discuss the state-of-the-art of generative techniques, share experience, consolidate successful techniques, and identify open issues for future work. The workshop was attended by a total of 22 participants, including the organizers, 2 invited speakers, presenters of 10 accepted paper contributions, and other ECOOP attendees. The call for participation, the contributions, and the workshop results can be accessed at www.generative-programming.org/ecoop2002-workshop.html.

The remaining part of this report is organized as follows. Section 2 gives an introduction to GP. The goals, topics, and the format of the workshop are outlined in Section 3. Sections 4 and 5 give a brief summary of each contribution including the debates. Section 6 gives an account of the identified future directions. The report ends with the list of participants and contributions.

2 Generative Programming and Technology Projections

Generative programming builds on system-family engineering (also referred to as product-line engineering) [5, 7] and puts its focus on maximizing the automation of application development [3, 2, 1, 4, 6]: given a system specification, a concrete system is generated based on a set of reusable components.

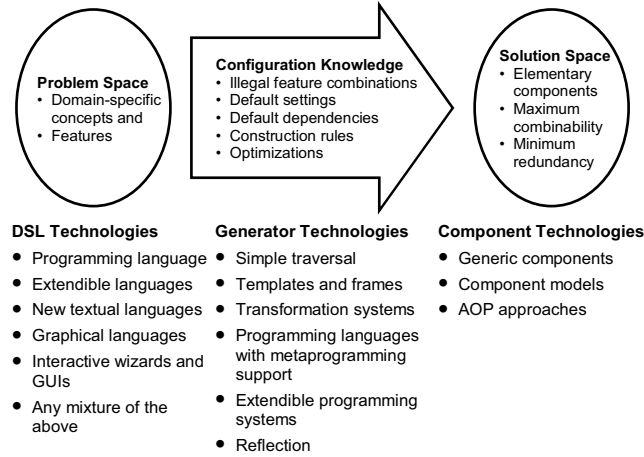


Fig. 1. Generative domain model and technology projections

The key to automating the assembly of systems is a generative domain model that consists of a problem space, a solution space, and the configuration knowledge mapping between them (see Figure 1). The solution space comprises the implementation components and the common system family architecture, defining all legal combinations of the implementation components. The problem space consists of the application-oriented concepts and features that application engineers use to express their needs. This space is implemented as a domain-specific language (DSL). The configuration knowledge specifies illegal feature combinations, default settings, default dependencies, construction rules, and optimizations. The configuration knowledge is implemented in the form of generators. The generated products may also contain non-software artifacts, such as test plans, manuals, tutorials, maintenance guidelines, etc.

Each of the elements of a generative domain model can be implemented using different technologies, which gives rise to different *technology projections*:

- *DSLs* can be implemented using programming language-specific features (such as in template metaprogramming in C++ [2]), extensible languages (which allow for domain-specific syntax extensions such as OpenC++, OpenJava [inv1], keyword-based programming [pos3], Refill [pos4], and Jasper [pos5]), graphical languages (e.g., UML, Simulink/Stateflow [pos6]), or interactive GUI and wizards (e.g., [pos9]).
- *Generators* can be implemented using template-based approaches (e.g., TL [3] and Jostraca [pos8]), built-in metaprogramming capabilities of a language (e.g., template metaprogramming in C++), transformation systems (e.g., QCoder [pos9]) partial evaluation (e.g., [pos12]), or extensible programming systems (e.g., OpenC++, OpenJava, keyword-based programming, Refill, and Jasper).

- *Components* can be implemented using, e.g., generic components (such as in the STL), component models (e.g., JavaBeans, EJB, ActiveX, or CORBA), or aspect-oriented programming approaches (e.g., AspectS [pos11]).

3 Workshop Topics and Format

Potential participants were asked to submit a two-page (or longer) position paper detailing their experience with GP, their perspective on the relation of GP and other emerging approaches, and their planned contribution to the workshop. Possible topics included

- synergy between object-oriented technology, components and generative techniques, styles of generators (application generators, generators based on XML technologies, template languages (e.g., JSP), template metaprogramming, transformational systems, intentional languages, aspects, subjects, etc), particularly their uses and limitations;
- design of APIs that support generative techniques
- generation of code artifacts, such as application logic, UIs, database schemas, and middleware integration;
- generation of non-code artifacts such as test cases, documentation, tutorials, and help systems;
- capturing configuration knowledge, for example, in DSLs, and extensible languages;
- influence of generative techniques on software architecture (e.g., building and customizing frameworks and applying patterns);
- testing generic and generative models; and
- industrial applications of generative technology.

The format of the workshop was designed to foster discussion and interaction rather than presentations. The workshop schedule consisted of two short presentation sessions in the morning, and one pro-and-contra session and one discussion session in the afternoon. The workshop started with a few words of explanation about the format and a short introductory talk on GP, with the contents outlined in Section 2. The short presentation sessions consisted of 10-minute talks by the authors of accepted paper contributions, followed by 5 minutes for answering questions.

Similar to last year, the short presentation sessions were followed by a pro-and-contra session. The pro-and-contra session consisted of a 10 minute pro-and-contra discussion for the six papers selected for the session. The idea was to have two volunteers for each paper, one defending the ideas presented in the paper and one trying to come up with counterarguments. Within the 10 minutes the pro and the contra volunteer were asked to make 2-minute initial statements and then to exchange their arguments. They were also allowed to ask the audience for additional arguments, if needed. The pro-and-contra session inspired lively debates and turned out to be a very efficient format to discuss the advantages and disadvantages of the different approaches and the relationships among them. The

workshop ended with a open discussion, in which we identified and summarized open issues and topics for future work.

4 A Summary of the Contributions and the Debates

The paper contributions to this workshop can be classified into three categories

- *language extension technologies*: OpenJava and OpenC++ [inv1], keyword-based programming [pos3], Refill [pos4], and Jasper [pos5];
- *other GP technologies*: Jostraca (a template-based approach) [pos8], QCoder (wizards and code transformations) [pos9], partial evaluation [pos12], and aspects [pos11];
- *applications of GP*: embedded software [pos6], graphical user interfaces [pos7], and distributed client-server systems [pos10]

The following subsections give a short summary of the contributions and the workshop debates.

4.1 Language Extension Technologies

OpenJava, OpenC++, and Javassist Shigeru Chiba was the first invited speaker and gave an overview of the tools OpenJava, OpenC++, and Javassist (see [inv1]). OpenJava and Open C++ are both source-to-source translation toolkits. The unique features are basically that compile-time reflection allows people to work with source-level vocabulary such as classes, methods, fields, types, etc. For typical programmers, source-level vocabulary is more familiar than parse-tree level stuff (such as declarators, initializers, etc.). As a consequence, OpenJava and Open C++ are easy to use for application-domain specialists, but also more limited than parse-tree level tools. As another consequence, both OpenJava and Open C++ have limited possibilities for syntax extension, again as an attempt to stay within the notion of standard source level vocabulary. For the method body, however, a direct parse-tree representation is used since creating (meta-) objects for each token is too complicated. However, note that this is not an AST (with a lot of BNF-non-terminals), but a concrete syntax tree. The API of both OpenJava and Open C++ is basically a Visitor visiting the parse/source tree: an event is raised when certain “events” occur when traversing the parse tree (such as a method call, instance creation, etc). The events can be limited to instances of specific classes and event listeners can be “attached” to specific classes. Javassist is a new tool that allows users to work with source-level vocabulary while internally modifying the byte code. Here there are of course no syntax extensions whatsoever.

Summary of the debate. Experience has shown that the idea of basing the translation on source level abstractions does indeed lower the initial learning curve for users of the system. Currently the three APIs of OpenJava and Javassist are somewhat different, due to historical reasons, but they could be more homogenized. It was discussed to what extent there could be a concept such as “compile-time reflection.” OpenJava and OpenC++ uses the word reflection to denote that the translation is at the source level rather than the parse tree level.

Keyword-Based Programming In [pos3], Thomas Cleenerck et al. present the notion of keyword-based programming. The goal is to devise a general mechanism for embedding domain specific language extensions into existing languages, or building entirely new ones. This requires a composable language. Keyword based programming allows named language constructs to be specified in terms of their translation into existing constructs, which can again contain keyword-based definitions. Keywords can be combined into languages through an XML specification. A simple example is the keyword `time`, which is used to time execution of statements, as in the following fragment:

```
...
    time {
        for(int I=0;I<100;I++) {
            stack.push(new Integer(i));
        }
    }
...

```

The `time` keyword can then be defined as

```
keyword time {
    AST {
        public class AST {
            keyword body;
        }
    }
    RecognitionPattern {
        "time", "{", keyword body, "}" )
    }
    generator {
        public JavaStatements generate() {
            JavaStatements js = generate body;
            >> long aXX = System.out.currentTimeMillis();
            >> $js;
            >> System.out.println("action has taken " +
                (System.currentTimeMillis() - aXX));
        }
    }
}

```

Summary of the debate. Several issues came up in the debate. One was the relation between keywords and syntactical domains (or, in other words, constructors and types/sorts) and whether there was a one to one mapping. The answer was that it was necessary to try to find the minimal set of keywords in order to avoid language explosion. The issue of conflicting keyword recognition patterns was raised, but dismissed as not being a problem in practice so far. It was stated that the keyword system was implemented to a large part in itself.

Refill Kasper Østerbye presented in [pos4] a framework for language extension based on the idea that it should be possible to extend not only syntax, but also static semantics, name binding rules, and compile time semantics. The focus is on providing a simple model for doing this, which like the two previous presentations do not force the language extender into parse theoretic issues, but enables the extender to stay at the host language level. Currently the system focuses on building an extensible version of Java. The idea is that using extended syntax should be as easy as using third party classes, as in the following example:

```
import mycontrolstructures.ForEach;
import java.util.Vector;
class Test {
    Vector employees = new Vector();

    public void printAllEmplees(){
        foreach Person p in employees
            System.out.println(p.name + " " + p.address);
    }
}
```

The import statement has the additional semantics that if the imported classes represent language extensions, the language is extended within this file. The language extensions are defined as slightly extended Java classes, which define compile-time actions such as name-binding, type check, etc. This approach, just as the one described previously, addresses the issue of a modular language design. This author also expressed the opinion that the problem of incompatible language extension has yet to prove itself as a practical issue, while it is of course an undecidable theoretic problem to see if two language extensions conflict even at the syntactic level, not to mention any of the semantic levels.

Summary of the debate. The issue came up how one could then extend the `foreach` construct in the example. Currently the approach is only suited to add new language constructs, not to modify existing ones. But language constructs can be places in hierarchies, and thus share common semantics, but concrete syntax cannot be inherited.

Jasper Dmitry Nizhegorodov presented in [pos5] another extensible Java compiler named Jasper. Jasper has been used extensively within the Oracle Corporation, both for EJB deployment, and for miscellaneous Java extensions, including multi-methods and generic functions. It is based on a compile-time meta-object protocol, and allows both syntax extensions as well as more traditional macro expansions. The tool is targeted at transformations, using template based code generation. The system is layered, with the bottom layer being an extensible parser and transformer. Both of these ensure that the abstract syntax tree is well formed at all times, through the use of Java's type system. On top of this, a macro layer has been built, which translates macro definitions into parser extensions. Here is an example:

```

MACRO Statement UNLESS (Expression test, Statement body) {
    return NEW Statement (Expression test, Statement body)
    if (test == false) body;
}

```

This translates into the two following definitions:

```

class UNLESS_Statement extends StatementParserExtension {...}
class UNLESS_StatementExpander extends StatementExpander {
    Expression test; Statement body;
    public Statement expand () {
        return
            new IfStatement
                (0,
                 new EqualExpression(0, test,
                                     new BooleanExpression(0, false)),
                 body,
                 null);
    }
}

```

The first is a class used for extending the parser, the second is the class used for defining the transformation. During parsing, the test and body fields of the statement expander class is set, and are hence available when the transformation tool calls the expand method. Syntactic validity is seen in that the return type of expand is a Statement, where the actual returned object is an if-statement, which is a subtype of statement.

Summary of the debate. The tool is completely implemented and mature, and it has been used to solve real world problems at Oracle. As such it was hard to argue with. Jasper only addresses transformations, not static semantic checks.

4.2 Other GP Technologies

Jostraca Richard Roger presented Jostraca [pos8]. Jostraca is a tool that focuses on the specification of code generation, for which purpose it uses a template-based approach inspired by Java Server Pages and XML. The main idea is that JSP and XML based techniques are existing tools for programmers today, and hence there is something to leverage on. A substantial part of the paper and presentation is devoted to reflection on the fact that generative programming is not mainstream, and hence to change this, it is important to investigate ways in which the average programmers can start to get some leverage today. And Jostraca is one such approach. A simple example is that of implementing getters and setters for fields in Java. In a JSP syntax, this could be implemented as:

```

<% String[] fields = new String[] { "FirstName", "LastName" }; int i; %>
public class Person {
    <% for( i = 0; i < fields.length; i++ ) { %>

```

```

        private String m<%=fields[i]%> = "";
        public String get<%=fields[i]%>() {
            return m <%=fields[i]%> ;
        }
        public void set<%=fields[i]%>(String p<%=fields[i]%>){
            m<%=fields[i]%> = p<%=fields[i]%>;
        }
    <% } %>
}

```

The idea is here that the fields are specified at top of the program, and the for-loop iterates over those fields, producing the code in the body of the for-loop for each field. Unfortunately, the syntax of such templates is somewhat clumsy. The paper discusses the readability issue and introduces an extra layer of syntactic sugar. Also, this new template can be run through the Java compiler to verify that it is at least syntactically correct, thus addressing an important issue, namely that of debugging templates.

Summary of the debate. The approach is best suited for specifying code generation, where there are a lot of repetitions for simple programs.

Wizards and code transformations The paper [pos9] by Marcelo d'Amorim et al. describes QCoder, a tool for the generation and maintenance of Java programs. The tool can be used to generate class structures including their implementation, and also to safely modify the new code by applying refactorings.

QCoder implements both code generation and refactoring using code transformations. Transformations are written in a language that extends Java with metaprogramming constructs, such as metavariables, semantic-based pattern matching, and meta-expressions. The language is specifically designed to write code transformations on Java programs. In contrast to a purely template-based approach such as Jostraca, the transformational approach of QCoder supports not only the instantiation of target templates, but also matching source templates on existing code to drive the instantiation of the target templates.

Code transformations in QCoder are performed by user-defined wizards. QCoder wizards can be easily defined by creating new or assembling existing wizards using a tool customizer. The composition of wizards involves composing graphical elements (implemented as JavaBeans) and code transformations.

An integration of QCoder with JBuilder and VisualAge for Java is available. The paper discusses the practical applications of the technology, e.g., introducing specific architectural patterns into an EJB application.

Summary of the debate. The first issue discussed by the audience was the problem of the after-the-fact modifications of code generated by wizards, which is known from popular IDEs. The problem is that by re-running a wizard with different option settings the modifications get lost. Although not fully eliminating the problem, the advantage of QCoder is that automatic refactoring can be applied to existing code, meaning that certain modifications corresponding to different option settings could still be achieved automatically. Another issue

discussed by the audience was the efficiency of code generation. In the case of QCoder, rather small transformations are performed in an interactive manner, meaning that the performance has not been a problem so far.

Aspects in Squeak Robert Hirschfeld [pos11] presented AspectS, an implementation of aspects for the Squeak language. According to the author, the design should also work for most other Smalltalk implementations. AspectS is built on top of John Brant’s implementation of method wrappers and extends the meta-object protocol to accommodate the aspect modularity mechanism. The implementation covers the aspect mechanisms known from AspectJ, including the cflow construct. Because of the Smalltalk architecture, aspects can be woven into an application and later removed, both at runtime. The paper discusses the implementation of AspectS in some detail and it gives examples of its use.

Summary of the debate. The discussion mainly revolved around clarifications. The performance is ok, as there is very low overhead in the method wrapper approach. The syntax for aspects are embedded in the usual Smalltalk syntax, and it uses a lot of blocks, and Smalltalk’s keyword-based method-names in order to somewhat improve the readability.

Partial Evaluation As exemplified by AspectS, dynamic reflection provides a flexible way to compose components. Unfortunately, reflective calls are considerably slower than their base-level counterparts. In [pos12], Susumu Yamazaki and Etsuya Shibayama present an approach to optimize reflective calls in Java using an automatic specialization technique. In contrast to previously proposed techniques, this approach is capable of specializing reflective calls to polymorphic methods and it also works in the context of dynamic loading.

The approach utilizes two techniques to reach this goal:

1. a specialization technique combined with binding time analysis
2. generative extension object technique (GEO)

The specialization technique can generate specialized versions of a reflective method for the subclasses of a given class. The reflective calls contained in such a specialized method are statically resolved based on the location of the specialized method in the class hierarchy.

The paper shows that, in the presence of dynamic loading, specialized methods sometimes need to be added to already loaded classes (e.g., in the case of double dispatching). This, however, is not directly possible without extending the JVM in a nonstandard way. As a solution to the problem, the paper proposes to use the extension object pattern, which can be statically added to a class requiring specialization. The extension object pattern provides a hook that allows extending the original object by object composition at runtime and thus, avoids the JVM extension problem.

In order to see the performance benefit of this technique, Yamazaki and Shibayama compared an unspecialized version of the reflective visitor pattern

with a specialized version with and without GEO (the specialization was performed by hand). The specialized version without GEO was about 100 times faster than the unspecialized version, and the one with GEO was 60 times faster than the unspecialized version. As of writing, a partial evaluator based on the described approach is under development.

4.3 Applications of Generative Programming

Embedded software In [pos6], Gerd Frick and Klaus Müller-Glaser make the point that embedded real-time systems is a real-world application domain where state-of-the-art software development processes are already generative. Visual formalisms (i. e., with graphical notations) have been used in control systems engineering for many years. For example, Statecharts are used to model discrete, typically reactive and open-loop control systems, and signal flow notations are used to model continuous, closed-loop control systems. Both kinds of notations can be used together to create hybrid models.

There is a large number of development environments supporting control-system design using the formalisms described above, e.g., Statemate, Matlab / Simulink / Stateflow, and ASCET-SD. Such development environments usually support system modeling, simulation, and code generation.

Generation of code for real-time embedded targets such as microcontrollers is becoming state of the art in the industry. The alternative term for code generation often used there is “autocoding.” Generated code starts being used in production systems, e.g., in automobiles. This became possible due to the availability of code generators that output highly optimized, production-quality code. The main advantage of this approach is the higher abstraction level. The executable system models correspond to programs written using DSLs.

Open issues are the integration of different modeling approaches as well as considerations of the physical model (e.g., distributed ECU architecture in the automobile). A first step is the use of XFL, an XML-based concept involving a metalanguage for representing languages and transformations, as the common mediator between existing languages and tools. XFL is currently under development at the FZI in Karlsruhe, Germany.

Graphical user interfaces In [pos7], Sofie Goderis and Wolfgang De Meuter propose to generate GUIs from high-level specifications by means of declarative metaprogramming. The goal of the proposal is to truly decouple GUIs and applications and to generate GUIs for different devices based on a single specification.

According to the authors, current GUI technologies do not sufficiently decouple GUI code from application code. For example, the Model-View-Controller pattern still requires change notification calls to be distributed in the application code. Furthermore, most user interface builders generate a GUI plus function stubs to be filled by application programmers, which requires manually changing the generated code or using subclassing. The authors would rather prefer a declarative way of coupling the user interface with the application code.

Currently, porting an application to a new device requires manually rewriting its GUI implementation. This could be avoided by generating the GUI for different devices (e.g., computer, mobile phone, PDA, etc.) from one specification. The idea is to write the specification in terms of higher level concepts, such as text, figure, listing, question, action, dependency, choice, etc., rather than the low-level concepts found in current GUI definition (e.g., button, field, frame, etc.).

The paper proposes to use Declarative Meta Programming (DMP) to generate GUIs by representing the high-level GUI specification and the necessary hooks in the application code as logic facts, and by using logic rules to express how to generate the code to combine both parts.

DMP can be done in SOUL, a Prolog-like programming language which is tightly integrated with Smalltalk. SOUL programs can reason about Smalltalk code and Smalltalk can call SOUL programs. The paper gives a simple example illustrating how SOUL could be used to solve the GUI generation problem.

The realization and validation of the described proposal is still in a very early stage.

Summary of the debate. The main issue discussed by the audience was the readability of logic programs. The problem of making the GUI specification more intuitive could be solved by providing a graphical tool on top of logical facts. However, the use of logical facts has also important advantages such as the possibility to represent domain knowledge in an explicit way and to use inference algorithms to check for conflicts. Another concern raised by the audience was the need to incorporate the potentially huge differences between GUI technologies of different devices. This problem could be addressed by using a layered model of device-independent or dependent rules. This approach can also help to achieve better reuse in the GUI domain. The issue of connecting a GUI to an application was also discussed. The Smalltalk meta-object protocol (which is supported by SOUL) was pointed out as a very flexible mechanism to connect a GUI to the application classes. Finally, the need to compare the effectiveness of the DMP generation approach should also be compared to other approaches, e.g., the template-based approach, was emphasized by the audience.

Distributed client-server systems In [pos10], Barrett Bryant et al. present UniFrame, a framework for integrating distributed heterogeneous components including functional and Quality of Service (QoS) constraints. UniFrame assumes that developers will provide on a network a variety of components for specific problem domains and the specifications of these components, which describe both their functionality and QoS. The components needed for a specific application will be retrieved by so-called “headhunters,” which identify sets of components that satisfy both the desired functionality and QoS from those available on the network.

After the component sets are identified and fetched, the distributed application is assembled by choosing one component from each set according to the generation rules embedded in a generative domain-specific model. This assembly

may require the creation of a glue/wrapper interface between various components as well as instrumentation to validate dynamic QoS parameters (e.g., response time). Once the system is assembled, it must be tested using a set of test cases supplied by the developers to verify that it meets the desired QoS criteria. If not, a different assembly of components is tried.

So-called Two-Level Grammars (TLG) (which originally have been introduced to specify Algol 68) are used to specify the generative rules and provide the formal framework for the generation of the glue/wrapper code. The two levels of a TLG are type definitions in the form of a context-free grammar, and function definitions in the form of another context-free grammar. TLG may be used to provide for attribute evaluation and transformation, syntax and semantics processing of languages, parsing, and code generation.

The paper gives the an example of automatically assembling a bank account management system from the client and server components for a bank domain that are available on the network.

Summary of the debate. One of the concerns raised was that glue code is typically something that is very special for a given context and therefore hard to reuse. In particular, it may be hard to predict, what glue codes the different components on a network may need. However, the main purpose of glue code in UniFrame is to adapt technologies, not component functionality, and a glue code generation capability for a set of standard technologies can be provided.

5 Collaboration in the GP Community

The second invited talk was by Joost Visser ([inv2]), who presented his initiative to provide the GP community with an efficient electronic collaboration platform. Given the growing interest for GP and the amount of workshops, conferences, tools, and companies related GP technology, there is a definitive need for a single, efficient, and persistent platform for collaboration in the GP community. The solution is a GPWiki site and a repository for collaborative software development.

GPWiki (www.program-transformation.org/gp/) is a collaborative web authoring for the GP community based on WikiWiki. WikiWiki is Hawaiian for “Fast!!” and it stands for a light-weight, easy-to-use system for collaborative web content management. WikiWiki allows you to edit pages via your browser. Any word with 2 non-consecutive capitals is a link. There are many implementations of WikiWiki. The one behind GPWiki is TWiki. GPWiki serves the GP community as a medium to summarize discussions (workshop debates, mailing lists); present approaches, methods, subfields; advertise tools, systems, languages, libraries; develop surveys, taxonomies; conduct comparative studies; grow bibliographies, glossary, indexes; maintain event calendar; vent opinions, ponder hunches, and outline perspectives.

The proposal for a platform for collaborative software development for the GP community is based on the idea that reuse is a catalyst for collaboration. This has been the experience from initiating and running such a platform for pro-

gram transformation tools at www.program-transformation.org/package-base/. The latter contains a collection of currently 57 packages for language prototyping, compiler construction, program transformation, software renovation, documentation generation, etc.

Joost identified two main ingredients of a collaborative software development platform for GP: a set of common exchange formats and an online component repository. A common exchange format would allow for the interoperability of various components, generators, front-ends, analyzers, back-ends, etc. A common format used for the program transformation component repository is ATerms. The online component repository needs to provide a common interface for building, testing, documenting, bundling, and distributing packages. The tool used by the program transformation component repository is autobundle.

The workshop participants were encouraged to contribute to GPWiki and the online repository.

6 Concluding Discussion

At the beginning of the discussion session, we came up with a prioritized list of discussion topics. The first issue that we discussed was correctness support for generative programming. Erik pointed out that we need typing mechanisms for language extensions and for manipulating languages. However, with many generative approaches, particularly the dynamic ones, the distinction between runtime and compile time becomes blurred. Thus, what is the role of strong typing? When do errors happen? Dmitry responded that in GP we also have things like macro-definition time, macro-expansion time, etc., but the goal should be to catch errors as early as possible. As another issue related to correctness, Kasper emphasized that the base language should be designed in a composable way in the first place. We need to think about what the useful, composable (orthogonal, compatible) features of programming languages are.

Language extensions should be checked statically, and have their own semantics and type system. However, the typing issue is tough because different extensions used in parallel might have influence on each other's semantics, errors and syntax. Jörg pointed out that we need a theory of extensible languages to reason about language extensions and compositions. An important observation made by Chiba was that the discussed correctness issues are not necessarily intrinsic to GP, but they also show up in the context of composing conventional libraries. However, correctness is not limited to type checking, as language extensions should cover editing, debugging, error checking, and code generation. Finally, Erik pointed that domain-specific language extensions can be used as a “wrapper” around complicated frameworks, allowing detecting framework usage errors at compile time instead of runtime. That's a big advantage of GP compared to conventional libraries and frameworks.

The second issue that was briefly discussed was the need for a comparison of different generative techniques. This would help researchers to better understand alternative approaches, and developers would certainly welcome more

guidance in selecting the appropriate technique for their problems. Such a comparison would be rather an ongoing effort that could be facilitated through GPWiki. In addition to a taxonomy and individual descriptions of approaches, the participants would welcome some good demonstrations of expressive power (or “expressive ease,” as Kasper pointed out) of individual approaches, common case studies, industrial success stories, and guidelines for selecting individual approaches.

The other issues identified at the beginning of the session, but not discussed for the lack of time, included the problem of customizing generated artifacts, reusability of language extensions, testing and debugging of generators, applications of GP, relations of GP to other disciplines, and feature interaction. Finally, at least half of the audience expressed their interest to participate in a follow-up, specialized workshop on extensible languages.

Participants

Joao Araujo (New University of Lisbon, Portugal; ja@di.fct.unl.pt), Uwe Bardey (University of Bonn, Germany; bardey@cs.bonn.edu), Paulo Borba (Qualiti Software Processes and Federal University of Pernambuco, Brasil; phmb@cin.ufpe.br), Barrett Bryant (University of Alabama at Birmingham, USA; bryant@cis.uab.edu), Shigeru Chiba (Tokyo Institute of Technology, Japan; chiba@acm.org), Thomas Cleenewerck (Catholic University of Leuven, Belgium; thomas@cs.kuleuven.ac.be), Krzysztof Czarnecki (DaimlerChrysler Research and Technology, Germany; czarnecki@acm.org), Erik Ernst (University of Aarhus, Denmark; eernst@daimi.au.dk), Gerd Frick (FZI Forschungszentrum Informatik, Karlsruhe, Germany, frick@fzi.de), Sofie Goderis (Free University of Brussels, Belgium; sgoderis@vub.ac.be), Robert Hirschfeld (DoCoMo Euro-Labs, Germany; hirschfeld@acm.org), Boris Magnusson (University of Lund, Sweden; boris@cs.lth.se), Dmitry Nizhegorodov (Oracle, USA; Dmitry.Nizhegorodov@oracle.com), Joost Noppen (University of Twente, The Netherlands; noppen@cs.utwente.nl), Kasper Østerbye (IT University, Denmark; kasper@it-c.dk), Richard Roger (InterComponentWare AG, Germany; Richard@Jostraca.org), Sybille Schupp (Rensselaer Polytechnic Institute, USA; schupp@cs.rpi.edu), Jrg Striegnitz (Research Center Jülich, Germany; J.Striegnitz@fz-juelich.de), Joost Visser (CWI, The Netherlands; Joost.Visser@cwi.nl), Markus Völter (independent consultant, Germany; voelter@acm.org), Jonathan Whittle (NASA Ames Research Center, USA; jonathw@ptolemy.arc.nasa.gov), Susumu Yamazaki (Tokyo Institute of Technology, Japan; syamazaki@acm.org)

List of Contributions

The invited presentations and workshop presentations and papers are available at www.generative-programming.org/ecoop2002-workshop.html.

1. Shigeru Chiba (Tokyo Institute of Technology, Japan), *Overview of OpenJava and OpenC++*. (invited talk)
2. Joost Visser (CWI, Amsterdam, The Netherlands), *Collaboration In the Generative Programming Universe* (invited talk)

3. Thomas Cleenewerck, K. Hendrickx, E. Duval, and H. Oliv   (Catholic University of Leuven, Belgium), *Capturing and using emergent knowledge by keyword based programming*
4. Kasper  sterbye (IT University, Denmark), *Refill - a generative Java dialect*
5. Dmitry Nizhegorodov (Oracle, USA), *Code-Generation Aspects of Jasper, a Reflective Meta-Programming and Source Transformations Processor*
6. Gerd Frick and Klaus D. M  ller-Glaser (FZI Forschungszentrum Informatik, Karlsruhe, Germany), *Generative development of embedded real-time systems*
7. Sofie Goderis and Wolfgang De Meuter (Free University of Brussels, Belgium), *Generating User Interfaces by means of Declarative Meta Programming*
8. Richard J. Rodger (InterComponentWare AG, Germany), *Jostraca: a template engine for generative programming*
9. Marcelo d'Amorim, Clovis Nogueira, Gustavo Santos, Adeline Souza, and Paulo Borba (Qualiti Software Processes and Federal University of Pernambuco, Brasil), *Integrating Code Generation and Refactoring*
10. Barrett R. Bryant, Fei Cao, Wei Zhao, Rajeev R. Rajee, Mikhail Auguston, Andrew M. Olson, and Carol C. Burt (University of Alabama at Birmingham, Indiana University Purdue University Indianapolis, New Mexico State University, USA), *Generative Programming Using Two-Level Grammar in UniFrame*
11. Robert Hirschfeld (DoCoMo Euro-Labs, Germany), *AspectS - Aspects in Squeak*
12. Susumu Yamazaki and Etsuya Shibayama (Tokyo Institute of Technology, Japan), *Runtime Code Generation for Bytecode Specialization of Reflective Java Programs*

References

- [1] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, October 1992, pp. 355-398
- [2] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000
- [3] J. C. Cleaveland. *Program Generators with XML and Java*. Prentice-Hall, XML Book Series, 2001
- [4] J. C. Cleaveland. Building Application Generators. In *IEEE Software*, vol. 9, no. 4, July 1988, pp. 25-33
- [5] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001
- [6] J. Neighbors. *Software construction using components*. Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine, 1980
- [7] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, 1999