Logic Metaprogramming Framework for Java

Ph.D. thesis

Logikai metaprogramozási keretrendszer Javához

Doktori (Ph.D.) értekezés tézisei

ESPÁK Miklós

Témavezető: Dr. FAZEKAS Gábor

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika és Számítástudományok Doktori Iskola

Debrecen, 2009

Contents

1 Aim and thesis of research project

My dissertation provides a solution to relieve difficulties of communication between Java and Prolog programming languages that arise from their semantic and structural differences. Applying declarative metaprogramming, I developed a flexible way of referring events within the program flow, and provided a naturalistic interface for describing these events. Thus, the aim of my research project –as well as its main result– is to create a logic metaprogramming framework for Java.

Software systems are extensively used in many fields of life today. These systems are usually extremely complex, and developing them needs a project team with high expertise in both software technology and the application domain. During the development, (among others) the business logic (concepts, processes, etc.) of the system have to be mapped to software artifacts. The mapping is not easy, since the systems of notions of the two fields are very different. On one hand, you have to extract the relevant concepts of the domain and reveal their relationships, which needs deep understanding of the field. On the other hand, you have to represent these concepts and their interrelationships with the means of the software model.

During a software development process, many models are developed from the high-level conceptual models to the low-level implementation models, describing numerous parts of the world being modelled. The main artifact produced in the course of the project is a computer program. Computer programs themselves can also be regarded as models, being a (hopefully valid) abstraction of the application domain.

Modelling a problem in a programming language requires you to think in the concepts of the language primarily, and to translate all the elements and relationships of the domain into the language. This requires much intuition, and we cannot expect that the process can be automatized ever. However, a programming language determines how one can think of a problem, and this is usually more constraining than the way you would express things in common sense. Moreover, the difficulty of modelling a particular problem in miscellaneous languages may be very different.

During the last half century, a myriad of *high-level* programming languages have been developed, and this is still an active field of research. The languages can be categorized into *programming paradigms* based on the target of the abstraction. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components [1]. They determine which aspects of the system appear in the program as individual program elements. The different programming paradigms require very different ways of thinking about the problem. None of them can be regarded

as superior to the other definitely. Actually, it depends on the type and complexity of the problem, which approach is the most suitable for it. There exist also *multiparadigm programming languages* that allow you to code various parts of the program along different paradigms.

Another obstacle of turning one's thoughts into a computer program easily is that programming languages differ very much from native languages both in their semantics and their syntax. (The semantics is closely related to the programming paradigm.) So, you have to learn to "speak" one or more programming languages "natively" so that you can write programs which are correct, concise, evolvable and reusable as well. In turn, there are *controlled natural languages* that are easy to understand and use even without a long learning curve. However, these are not used in general purpose programming languages now.

2 Technical background

2.1 Declarative metaprogramming

The project underlying this thesis was mostly inspired by *declarative metaprogramming* (DMP). Declarative metaprogramming is defined as the use of a declarative programming language for writing metaprograms [12]. Declarative languages are very suitable for writing metaprograms because they allow the programmer to focus on what to achieve rather than how to achieve it. One typical task of metaprograms is to look up parts of the program being processed. Using the imperative approach this "search" could be done by navigating through metaobjects representing the elements of the program, examining one by one whether the search criteria applies for them. However, this path can change as the program evolves, which can result in the metaprogram having to be adopted to it. The quest would be easier if you could define *"what"* you are looking for separately from *"how to"*, which is a basic idea of logic programming.

The most natural way to organize *metaclasses* of a language (classes of *metaobjects* storing metainformation) is a hierarchy [10]. Although metaobjects can be represented as Prolog terms, but –because of the lack of support for inheritance– there is no object polymorphism between terms representing metaobjects. A class hierarchy definition would allow that rules which accept terms (objects) of a class, also accept objects of any subclass of the class automatically.

2.2 Aspect-oriented programming

One of the key issues of designing computer programs is called *separation of concerns* (SoC). It expresses that the program should be modularized in such a way that program elements representing the same *concern* of the problem should be placed in the same module, and one module should represent only one concern in an ideal case.

The aim of *aspect-oriented programming* is to separate such *crosscutting concerns*. To achieve this it suggests to extract the scattered code representing a single concern into a distinct module. This special module is usually called *aspect*. Additionally, you have to specify the places where this separated code and the original one (cleaned up from the tangled concerns) should be joined together. The moments of execution when the control is passed over between the code of concerns are called *join points*. The set of join points can be defined by *pointcuts*. Aspects may have special subprograms called *advices*, which cannot be invoked directly, but are activated at the join point. This is called *implicit call*. In contrast with ordinary subprograms, the places of the invocation of advices are specified by the advices themselves by *pointcuts*.

It is a very critical issue of AOP systems that how precise the set of the join points can be described. As you have to designate the points in the program execution, it is obviously an issue of metaprogramming. It has been shown that simply enumerating places in the code causes that the pointcut will be *fragile*, which means that refactoring the base code involves that the pointcut designator has to be modified as well [9]. To enhance the reusability of aspects, pointcuts have to be described in a more robust way, by the properties of the join points or by their relationships. Logic metaprogramming –as discussed above– provides powerful techniques for this.

Another critical issue of aspect-oriented systems is when weaving is performed. Compile-time AOP systems require the source code of both the aspects and the base program to compose them by weaving producing either source files (precompilation) or object files. In general, weaving at a later time makes the system more dynamic and flexible. Load-time or run-time weaving allows you to weave (or even *unweave*) aspects for classes which were not available at the time of launching the application (possibly loaded through the network), or just whose source code is not available. Although load-time and run-time weaving causes some time overhead since the object code has to be instrumented during the execution, weaving is usually done only once during the life-cycle of a class. I have further demonstrated that run-time weaving does not decrease the efficiency of the composed program inevitably but it can even improve that [3].

2.3 Controlled natural languages

Controlled natural languages (CNL) are not a programming paradigm, but only a class of languages. They are subsets of natural languages with a restricted vocabulary and a narrow set of grammar rules. Controlled language texts can be read just as easily as natural languages, but they are unambiguous, which is an important factor for processing them by computers.

Controlled natural languages are used successfully and actively in industry for specific application domains [15]. Also, there are standards of controlled subsets of English for decades [14, 5]. Another excellent example for CNLs, is the translation of the OMG's Semantics of Business Vocabulary and Business Rules into English [7]. Although the "official" definition is in XML, the appendices contain it in a controlled language form, which is much easier to read for humans.

Lopes et al. suggest to use a controlled natural language for writing general purpose programs [11]. This approach raises some concerns, though. Rewriting an algorithm into this language, preserving the original structure of the code, results in a very verbose text, which would rather decrease the readability of the program instead of improving it. Lopez et al. suggest using anaphoras to eliminate local variables, which would result a code being concise and quite easy to read, especially for domain experts not familiar with programming languages.

However, CNLs are popular in specific application domains. Regarding our field, metaprogramming could be considered as such an application domain. As already discussed, metaprogramming can be done best in a declarative way, which is an inherent property of natural languages. Moreover, in aspect-oriented programming there is a special, very restricted use of metaprogramming for specifying pointcuts. Even, the structure of pointcut designators is quite simple, which would make it relatively easy to define a controlled natural language interface upon them.

3 New results

3.1 Multiparadigm programming

An ideal multiparadigm programming framework or language should allow intermixing the constructs of the languages without forcing the programmer to deal with the internal representation of terms in the system. I developed the Prolog4J multiparadigm programming framework, in which this internal representation remains completely hidden from the programmer. The framework integrates Prolog into Java. Additionally, I presented a tuProlog library

that allows object-oriented style programming in Prolog. As a third result in this topic, I also introduced a lingual symbiosis of the Java and Prolog language, which I named Japlo [4].

3.1.1 Prolog4J

In contrast with foreign language interfaces (like JPL [16] for SWI-Prolog [17]) and Prolog engines written in Java (like tuProlog [2] or JLog [8]), the Prolog4J framework exploits the advantages of Java 5. In this new generation of the Java platform many new language features were introduced which allows to build a more easy-to-use and type-safe interface over Prolog programs.

In Prolog4J a Prolog query produces a `Solution<A>` object, through which the results can be accessed. The `Solution<A>` class implements `java.lang.Iterable<A>` so that the individual solutions can be traversed by a for-each loop easily. If you are interested in the binding of another variable of the goal, other iterable objects can be achieved through the result of the query. There are helper methods in `Solution<A>` as well for collecting the solutions into specific Java collections.

Using generics and autoboxing allows the programmer to get rid of type casts and wrapping primitive values to objects or vice versa. There is also a well-defined way of converting Java objects to Prolog terms. Java objects are directly translated into tuProlog terms and vice versa. This is in contrast with P@J, which provides an additional type system over tuProlog to introduce type safety with generics. The P@J type system is a bridge between Java objects and Prolog terms, which doubles the number of conversions required and also needs more memory since an intermediate representation has also to be stored. Another advantage of Prolog4J is that the programmer can use Java types directly, so he/she does not have to deal with an intermediate type system at all.

The Prolog4J framework uses the Metadata Facility of Java 5 to specify an interface for a Prolog query. Such an interface is a method annotated by `@Goal`. The query can be specified as the argument of `@Goal`, and the variables of the goal can be bound to the formal arguments or the return value of the method by annotations as well (`@In`, `@InOut` or `@Out`). From the appearance of JDK 7, `@NonNull` annotations will be allowed to denote ground arguments, which will allow using these methods in a more type-safe way. Fig. 1 illustrates the use of Prolog4J.

Another innovative feature of Prolog4J is that the body of goal methods are generated automatically, using the Java Compiler API [13]. This feature works only with the Sun's Java compiler, however.

```
@Theory({
  "remove([X|Xs],X,Xs).",
  "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
  "permutation([],[]).",
  "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs), permutation(Zs,Ys)."})
public class PermutationTest {
  @Goal("permutation(X, Y).")
  static @Out("Y") Solution<List<Integer>> perms(
                                    @In("X") List<Integer> list){
    throw null;
  }
  public static void main(String[] args) {
     List<Integer> list = Arrays.asList(1, 2, 3);
     for (List<Integer> li: perms(list))
       System.out.println(Collections.max(li));
  }
}
```

Figure 1: Defining goal methods in Prolog4J

3.1.2 OOLibrary

OOLibrary is a tuProlog library provided as part of the Prolog4J framework. The library has dual purpose. Its main goal is to allow object-oriented style programming in Prolog. The library defines a coherent type system. Over the atomic types of Prolog, the type of compound terms is defined, there are list types and generic types as well. New types can also be defined.

OOLibrary allows you to define a hierarchy of classes. Classes can have fields, and they can have one superclass (single inheritance). The purpose of this hierarchy is to introduce *object polymorphism* into Prolog: although the instances of classes are ordinary (compound) terms, the instance of a subclass can stand for an instance of one of its superclass at any place of a program.

Methods can also be defined. They are not part of the class definition. The type of their formal arguments are specified in the formal argument list. The type of the actual arguments is checked dynamically at the application of the method.

The second purpose of the library is to improve the integration between Prolog and Java. If a term which is an instance of an OOLibrary class has to be converted to Java, Prolog4J will construct an instance of the corresponding Java class annotated by `@Term`. Similarly, the objects of these classes can be converted to OOLibrary objects. This improves the integration of Java and Prolog significantly, since the same class hierarchy of the application domain can be used in both languages, and the type conversion between them happens automatically.

3.1.3 Japlo

The Japlo language [4] provides an alternative way of Java-Prolog integration, which is one of my first results in the field. The language is an extension of Java, in which Prolog rules can be expressed with a Java-like syntax. The syntax was constructed in such a way that it should be easy to learn for programmers experienced in Java but not in Prolog.

Many of the concepts of Java appear in the way you can formulate rules. For example, a few control structures of Java can be used (if, while, do-while), rules can have return types, like methods, and the application of such rules are expressions. On the other side, the application of rules from Java methods is also easy: at first, there is a special operator (unary `+`) for testing the existence of a solution, at second, the solutions of a rule application can be iterated through by a for-each loop.

The syntax of Japlo provides a convenient way of writing rules to be accessed from Java. I showed that a Japlo rule can be applied in a much more concise way than JLog, if some Java objects have to be passed to it. Another advantage of Japlo is that it provides static type checking even for rules.

3.2 Logic metaprogramming framework

I developed a logic metaprogramming framework for Java. The framework is based on OOLibrary and Prolog4J. It has two layers. The low-level layer stores the metainformation of the program as facts (meta-database). The high-level layer defines OOLibrary classes and methods for examining the contents of the database.

Naturally, new methods can be defined over the existing ones, freely. So, logic-based analysis of the program is possible. Fig. 2 shows a simple example for this. The `isSingleton` method determines whether the Singleton pattern ([6]) is applied for a class or not. If the class is singleton, the method and the field which take part in the pattern will be bound to the arguments of `isSingleton`.

To retrieve the required metainformation, only a query has to be formulated, which specifies the properties of the required data and logical relationships between them. Within the framework the body of methods can be achieved in the form of an OOLibrary object so that they can be transformed with the means of Prolog. Moreover, the metainformation can be extracted from the bytecode, not the source, which allows to use the framework in a dynamic way.

```
isSingleton(C: classSpec, M: methodSpec, F: fieldSpec) :-
  methodOf(C, M), isPublic(M), isStatic(M),
  returnTypeOf(M, CDesc), descriptorOf(C, CDesc), argTypesOf(M,[]),
  fieldOf(C, F), isPrivate(F), isStatic(F), descriptorOf(F, CDesc),
  reads(M, F),
  methodOf(C, Ctr), isConstructor(Ctr),
  (not(isPrivate(Ctr)) -> !, fail).
```

Figure 2: The isSingleton method

3.3 Aspect-oriented programming framework

I developed an *aspect-oriented programming* framework upon the aforementioned DMP framework. It makes it possible for the programmer to refer to events of the execution of a program and makes certain subprograms run automatically when these events occur (*implicit invocation*). The descriptions that pick out the points of the program flow (called *pointcuts*) can be formulated as Prolog queries on the code. Besides of the predicates exploring the program code, spatial and temporal relations between events can also be expressed. Pointcuts can be specified as annotations on methods.

I showed that the expressivity of the pointcuts is high for two reasons. At first, since pointcuts can refer to the program elements by metavariables and the properties of these elements and their relationships can be examined through the DMP framework. At second, since temporal relationships between the events can be expressed.

The use of metavariables increases the robustness of pointcuts in terms of the program evolution because it allows to describe the pointcuts by their semantics instead of their syntactic appearance in the code.

Another advantage of my AOP system is that it is based on a mainstream programming language, not a prototype language. Moreover, aspects are weaved into the bytecode of classes, not their source. This made it possible to weave aspects at load-time. Using the Java Debugger Interface (JDI), run-time weaving is also possible. I showed in [3] that in certain cases run-time weaving does not decrease the speed of a program but it may even improve it.

3.4 Controlled natural language interface

I developed a controlled natural language interface over the pointcut definition language of the AOP system. Controlled natural languages are perfectly suitable for defining pointcuts, for several reasons:

Language	*Pointcut*	*Length*
AspectJ	set(FigureElement.x) || set(FigureElement.y) || set(FigureElement.start) || set(FigureElement.end)	98
LogicAJ	(set(?T.x) || set(?T.y) || set(?T.start) || set(?T.end)) && equals(?T, FigureElement)	85
CNL interface	changing x, y, start or end of a 'FigureElement'	48

Table 1: The length of equivalent pointcut definitions

- They describe a specific domain. Here, the domain is not the application of the program but metaprogramming as such.
- Pointcut designators have well-defined, simple structure, which provides two benefits:
  - The programmer will probably not constrained of the rigid structure of the controlled language.
  - The language can be constructed quite easily. (An implementation issue only.)
- Pointcuts should be defined in a declarative way for avoiding fragility, and declarations are the most basic form of communication in natural languages.

Although natural language texts are typically more verbose than formal descriptions, I built several grammatical techniques in the language that keep pointcut definitions as short as possible. I showed that the CNL formalism is more concise than the formalism of AspectJ or ALPHA. A specific example is given in Table 1.

Although natural languages texts are inherently ambiguous, this is (typically) not true for controlled natural languages. The grammar of the language I proposed does not allow any ambiguity.

A more sophisticated example is shown in Table 2. This example illustrates some other advantages of the CNL interface, which are much more essential than conciseness. At first, CNL pointcut definitions are easy to read or write even for those who are not familiar with aspect-oriented programming at all. At second, it hides the internal use of Prolog definitively, saving the programmer learning and using a second programming language in his/her program.

ALPHA	*Controlled natural language*
set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, this.d, 'drawAll', _)), cflow(T1, T2), reachable(Q, P), instanceof(Q, 'FigureElement')	changing F of P, calling drawAll of d last time at T, reading F of P during T, P is reachable from a 'FigureElement'

Table 2: The "cflowreach" pointcut in ALPHA and CNL

4 Summary

Summarizing the results of the dissertation it can be declared that the Java/Prolog multiparadigm programming framework (Prolog4J) that I have developed is easier to use than other similar projects in this field for several reasons. At first, it exploits the new language features of Java 5 like for-each loop, autoboxing or annotations. At second, it performs type conversions between Java objects and Prolog terms automatically, even for complex data types, without introducing new (wrapper) types. Finally, because it allows defining interfaces for Prolog queries as Java methods whose body can be generated automatically.

I also developed a declarative metaprogramming framework that makes it possible to reason about Java programs. The DMP framework is based on processing of the bytecode of Java classes, which makes it an ideal foundation for a dynamic aspect-oriented programming system.

The AOP framework I created has several benefits against current AOP systems. At first, it supports the use of metavariables, using which pointcuts can be expressed in a much more robust way. At second, temporal relationships between join points can be expressed. Another advantage is that aspects can be weaved dynamically, at load-time or run-time. Although some of these features appeared in other AOP systems, none of them supports each of them.

Finally, I developed a controlled natural language interface for defining pointcuts. This is a novel approach, which improves the readability of pointcuts significantly and makes it possible to formulate pointcut definitions even for programmers not experienced in aspect-oriented programming at all. CNL pointcuts are not only easy-to-read but usually even shorter than equivalent pointcuts in other AOP languages.

1 Kutatási cél és témameghatározás

Az értekezésem megoldást kínál a Java és Prolog programozási nyelvek közötti szemantikai és strukturális különbségekből adódó programozási nehézségek könnyítésére. A deklaratív metaprogramozás alkalmazásával kifejlesztettem a program futása során bekövetkező események kijelölésének egy rugalmas módját, és egy természetes nyelvhez hasonló felületet biztosítottam ilyen események leírására. Kutatói munkám célja és egyben eredménye egy logikai metaprogramozási keretrendszer létrehozása Javához.

A szoftverrendszerek manapság az élet számos területén meghatározó szerepet töltenek be. Ezek a rendszerek jellemzően igen összetettek, kifejlesztésükhöz olyan munkacsoportokra van szükség, amelynek tagjai magas szakértelemmel rendelkeznek mind a szoftvertechnológia, mind az alkalmazás szakterülete terén. A fejlesztés folyamán le kell képezni (többek között) a rendszer üzleti logikáját szoftver elemekre. A leképezés nem könnyű feladat, hiszen a két terület fogalmi rendszere igen különböző. Egyfelől ki kell emelni a szakterület lényeges fogalmait, és fel kell tárni a köztük rejlő összefüggéseket, ami alapos ismereteket igényel az adott területen. Másfelől, ezeket a fogalmakat és kapcsolataikat meg kell jeleníteni a szoftvermodell eszközeivel.

A szoftverfejlesztési folyamat során számos modellt ki kell fejleszteni a magas szintű koncepcionális modellektől egészen az alacsony szintű implementációs modellekig, a modellezett világ különböző részeit leírva. A projekt során előállított legfőbb termék a számítógépes program. A programok maguk is modelleknek tekinthetők, melyek az alkalmazás szakterületének (remélhetőleg helyes) absztrakciói.

Ahhoz, hogy egy problémát egy programozási nyelven modellezzünk, a nyelv fogalmi rendszerében kell gondolkodnunk, és a szakterület elemeit és azok kapcsolatait a nyelv eszközeire kell átalakítanunk. Ez magas fokú intuíciót igényel, és nem számíthatunk arra, hogy a folyamat bármikor is teljesen automatizálható lesz. Viszont a programozási nyelvek meghatározza, hogyan gondolkodhatunk egy problémáról, és ez rendszerint erősen korlátozott mód ahhoz képest, ahogy ezeket a dolgokat magunktól kifejeznénk. Ráadásul ugyanazon probléma modellezésének a nehézsége a különböző nyelvekben teljesen más lehet.

Az elmúlt fél évszázad során számtalan *magas szintű* programozási nyelvet fejlesztettek ki, és ez ma is aktív kutatás terület. Ezeket a nyelveket *programozási paradigmákba* sorolják aszerint, hogy az absztrakció mit céloz meg elsősorban. A paradigma olyan alapelvet jelent, amely alapján egy problémát értelmezni lehet és kezelhető összetevőkre lehet bontani [1]. Meghatározzák, hogy a rendszer mely vonatkozásai jelennek meg a programban külön programelemekként. A különböző programozási paradigmák gyökeresen

eltérő gondolkodási módot követelnek meg. Egyik sem tekinthető viszont egyértelműen felsőbb rendűnek a többinél. Valójában a probléma jellege és bonyolultsága határozza meg, melyik megközelítés a leginkább alkalmas a leírására. Léteznek *többparadigmás nyelvek* is, amelyek megengedik, hogy a program különböző részeit más-más paradigma alapján kódoljuk le.

Egy másik akadálya annak, hogy gondolatainkat könnyedén egy programozás nyelvre ültessük át az, hogy a programozási nyelvek rendkívül különböznek a természetes nyelvektől mind felépítésüket, mind szemantikájukat tekintve. (A szemantika szorosan kötődik az adott programozási paradigmához.) Tehát lényegében „anyanyelvi szinten" kell beszélni egy vagy több programozási nyelvet ahhoz, hogy olyan programokat írhassunk, amelyek egyaránt helyesek, tömörek, továbbfejleszthetők, újrafelhasználhatók, hatékonyak és így tovább. Ezzel szemben léteznek ún. *kontrollált természetes nyelvek*, amelyek könnyen érthetők és használhatók hosszadalmas tanulási folyamat híján is. Ezeket viszont nem használják általános célú programozási nyelvekben jelenleg.

2 Technikai alapok

2.1 Deklaratív metaprogramozás

Az értekezés során elkészített projektet leginkább a *deklaratív metaprogramozás* (DMP) inspirálta. A deklaratív metaprogramozás deklaratív programozási nyelv használatát jelenti metaprogram írása céljából [12]. A deklaratív nyelvek igen alkalmasak metaprogramozásra, mivel használatukkal a programozó arra összpontosíthat, *amit* el akar érni, nem pedig arra, *ahogy* el lehet érni azt. Metaprogram írása során tipikus feladat a feldolgozás alatt álló program bizonyos részeinek megkeresése. Az imperatív megközelítés szerint ez a "keresés" úgy végezhető el, ha a program elemeit reprezentáló meteobjektumok asszociációs láncolatán keresztül navigálunk, egyenként megvizsgálva az elemeket, megfelelnek-e a keresési feltételnek vagy sem. Ez a lánc viszont változhat, ahogy fejlődik a program, aminek következtében a metaprogramot is a változásokhoz kell igazítani. A keresést megkönnyítené, ha azt tudnánk megadni, hogy „mit" keresünk, nem pedig azt, „ahogy", ami a deklaratív programozás alapeszméje.

Egy nyelv *metaosztályai* (a program metainformációit tároló *metaobjektumok* osztályai) rendszerezésének legtermészetesebb módja egy hierarchia kialakítása közöttük [10]. Bár a metaobjektumok ábrázolhatók Prolog termek formájában, viszont – az öröklődés hiánya okán – nincs helyettesíthetőség a metaobjektumokat reprezentáló termek között. Egy

osztályhierarchia-definíció lehetővé tenné, hogy azok a szabályok, amelyek egy adott osztályba tartozó metaobjektumokat elfogadnak, elfogadják az alosztályok objektumait is.

2.2 Aspektusorientált programozás

Számítógépes programok tervezésének egy kulcskérdése a program egyes *vonatkozásainak elkülönítése*. Ez azt fejezi, ki, hogy a programot úgy kell részekre bontani, hogy a probléma azonos *vonatkozását* képviselő programelemeket azonos modulba kell helyezni, és ideális esetben egy modulban csak egy vonatkozás kódja jelenik meg.

Az *aspektusorientált programozás* (AOP) célja a több modult *átszövő vonatkozások* szétválasztása. Ennek elérésére azt javasolja, hogy az azonos vonatkozást képviselő, de szétszóródott kódot emeljük ki külön modulba. Ezt a speciális modult rendszering *aspektusnak* hívják. Emellett le kell írni azokat a helyeket, ahol az elkülönített kódot és az eredetit (amit a belekeveredett egyéb vonatkozásoktól megtisztítottunk) össze kell kapcsolni. A végrehajtásnak azon pontjait, ahol a vezérlés átadódik az egyes vonatkozások kódjai között, *csatlakozási pontoknak* nevezzük. A csatlakozási pontok halmaza *vágásponttal* írható le. Az aspektusoknak lehetnek speciális alprogramjai, ún. *javaslatok*, amelyek közvetlenül nem hívhatók meg, de a csatlakozási pontokban aktiválódhatnak. Ezt *implicit hívásnak* nevezzük. A közönséges alprogramokkal szemben a javaslatok hívásainak helyét maguk a javaslatok jelölik ki vágáspont megadásával.

Az AOP rendszerek nagyon kritikus kérdése, hogy milyen pontosan tudjuk leírni a kérdéses csatlakozási pontok halmazát. Mivel a program végrehajtásának pontjait kell kijelölni, ez nyilvánvalóan metaprogramozási kérdés. Kellens és mások megmutatták, hogy a program pontjainak egyszerű felsorolása a vágáspontot *törékennyé* teszi [9]. Ahhoz, hogy az aspektusok újrafelhasználhatóságát növeljük, a vágáspontokat robusztusabb módon kell megadnunk, azok tulajdonságai és kapcsolatai alapján. A logikai metaprogramozás – ahogy azt az imént tárgyaltam – erőteljes eszközt ad ehhez.

Az AOP rendszerek másik sarkalatos kérdése, hogy mikor történik a szövés. A fordítási idejű AOP rendszereknek mind az alapprogram, mind az aspektusok forráskódjára szükségük van ahhoz, hogy össze tudják építeni őket. Általánosságban igaz, hogy a későbbi fázisban elvégzett szövés növeli a rendszer dinamikusságát és rugalmasságát. A betöltési és a futási idejű szövés lehetővé teszi aspektusok beépítését (vagy éppen eltávolítását) olyan osztályokba is, amelyek nem álltak rendelkezésre a program indításakor (például mert hálózaton keresztül töltődtek be). Bár a betöltési és a futási idejű szövés jelent némi többlet terhet, mivel a kódot futás közben kell módosí-

tani, a szövés rendszerint egyetlen alkalommal történik meg egy osztály élettartama során. Továbbá megmutattam, hogy a futási idejű szövés nem csak hogy nem feltétlenül rontja egy program teljesítményét a betöltési idejű technikához képest, de akár javíthatja is azt [3].

2.3 Kontrollált természetes nyelvek

A *kontrollált természetes nyelvek* (CNL) nem jelentenek programozási paradigmát, csupán nyelvek egy kategóriáját. Ezek a természetes nyelvek részhalmazai, korlátozott szókinccsel és nyelvtani szabályok egy szűk halmazára. A kontrollált nyelvű szövegek csaknem olyan könnyen olvashatók, mint a természetes nyelvűek, viszont egyértelműek, ami igen fontos szempont a számítógépes feldolgozás szempontjából.

A kontrollált természetes nyelveket sikeresen alkalmazzák az iparban speciális szakterületeken belül [15]. Az angol nyelv kontrollált részhalmazára évtizedek óta létezik szabvány [14, 5]. A CNL-ek használatára másik kiváló példa az OMG üzleti szótár és szabálygyűjtemény szemantikájának az angol nyelvre ültetése [7]. Bár a „hivatalos" definíció XML-ben van megadva, a függelék tartalmazza annak kontrollált nyelvi formáját is, amely sokkal olvashatóbb emberek számára.

Lopes és mások kontrollált természetes nyelv használatát javasolják általános célú programok esetén is [11]. Ez a megközelítés viszont felvet néhány problémát. Ha az eredeti algoritmust úgy ültetjük át erre a nyelvre, hogy megőrizzük annak a szerkezetét, az igen bőbeszédű kódot eredményez, ami inkább rontja az olvashatóságot mintsem javítaná. Lopez és mások ezért utalószavak használatát javasolják a lokális változók kiküszöbölésére, amely viszonylag tömör és jól olvasható megfogalmazást eredményez, különösen olyanok szakterületi szakértők számára, akik nem jártasak a programozási nyelvekben.

Ugyanakkor a CNL nyelvek leginkább konkrét alkalmazási szakterületeken népszerűek. Az értekezés témakörét illetően a metaprogramozás is ilyen területnek tekinthető. Amint már tárgyaltam, a metaprogramozás leghatékonyabban deklaratív módon művelhető. A deklaratív szó kijelentéseket takar, ami a természetes nyelvek legalapvetőbb közlési formája. Ezen felül az aspektusorientált programokban a metaprogramozás igen korlátozott módon jelenik meg, méghozzá a vágáspontok megadásában. Ráadásul a vágáspontok leírásának szerkezete meglehetősen egyszerű, amely viszonylag könnyűvé teszi egy természetes nyelvi felület kialakítását felettük.

3 Új eredmények

3.1 Többparadigmás programozás

Egy ideális Prolog/Java többparadigmás keretrendszernek vagy nyelvnek anélkül kell lehetővé tenni a nyelvi konstrukciók kombinálását, hogy a programozót arra kényszerítenék, hogy a termek belső reprezentációjával foglalkoznia kellene. Kifejlesztettem a Prolog4J többparadigmás programozási keretrendszert, amelyben ez a belső reprezentáció a programozó elől teljesen rejtve marad. A keretrendszer a Prolog és Java nyelvek integrációját biztosítja. Ennek részeként elkészítettem egy tuProlog programkönyvtárat, amely objektumorientált stílusú programozást tesz lehetővé Prolog programokon belül. A témakör harmadik eredményeként bemutattam a Japlo nyelvet, amely a Java és Prolog nyelvi szimbiózisa [4].

3.1.1 Prolog4J

A külső nyelvi felületekkel (amilyen pl. a JPL [16, 17]) és a Java nyelven írt Prolog motorokkal (mint a tuProlog [2] vagy a JLog [8]) szemben a Prolog4J kihasználja a Java 5 előnyeit. A Java platform eme új változatában számos olyan új képességet vezettek be, amely lehetővé teszi könnyebben olvasható és típusbiztosabb felület kialakítását Prolog programok fölé.

Prolog4J-ben egy lekérdezés egy `Solution<A>` objektumot szolgáltat, amelyen keresztül az eredmények elérhetők. A `Solution<A>` osztály implementálja az `Iterable<A>` interfészt, így a megoldások könnyedén, egy foreach ciklus segítségével bejárhatók. Ha a cél egy másik változójának a kötései érdekelnek minket, a kapott objektumból egyéb iterálható objektumokat is megkaphatunk. A `Solution<A>` osztály segédmetódusokat is tartalmaz, amelyekkel a megoldások különböző kollekciókba gyűjthetők.

A generikusok és az automatikus csomagolás használatával a programozó megszabadul a típuskonverzióktól, és a primitív értékek, illetve csomagoló objektumok közötti oda-vissza átalakításoktól. A Java objektumok Prolog termmé alakításának jól definiált módja van. A Java objektumok közvetlenül tuProlog termekké lesznek konvertálva és viszont. Ez másképp van a P@J keretrendszer esetén, amely egy új, köztes típusrendszert biztosít a tuProlog típusai fölé azért, hogy a generikusok segítségével nagyobb típusbiztonságot érjen el. A P@J típusrendszere tehát egy hidat képez a Java objektumok és a Prolog termek között, ami megduplázza a szükséges konverziók számát, és némi többlet tárat is igényel, hiszen a köztes reprezentációt szintén tárolni kell. A Prolog4J megvalósításnak további előnye, hogy a programozó közvetlenül Java típusokat használhat, így nem kell foglalkoznia semmiféle

```
@Theory({
  "remove([X|Xs],X,Xs).",
  "remove([X|Xs],E,[X|Ys]):-remove(Xs,E,Ys).",
  "permutation([],[]).",
  "permutation(Xs,[X|Ys]):-remove(Xs,X,Zs), permutation(Zs,Ys).")
public class PermutationTest {
  @Goal("permutation(X, Y).")
  static @Out("Y") Solution<List<Integer>> perms(
                                    @In("X") List<Integer> list){
    throw null;
  }
  public static void main(String[] args) {
     List<Integer> list = Arrays.asList(1, 2, 3);
     for (List<Integer> li: perms(list))
       System.out.println(Collections.max(li));
  }
}
```

Figure 3: Cél metódusok megadása Prolog4J-ben

köztes típusrendszerrel egyáltalán.

A Prolog4J keretrendszer kihasználja a Java 5 metaadat-kezelő képességét a Prolog lekérdezésekhez programozói felület megadására. Az ilyen felületet `@Goal` annotációval ellátott metódusok jelentik. A lekérdezés az annotáció paramétereként adható meg, és a lekérdezés változói szintén annotációk segítségével rendelhetők a formális paraméterekhez, illetve visszatérési értékhez (`@In`, `@InOut` vagy `@Out`). A JDK 7 megjelenésétől kezdve a `@NonNull` annotáció is használható lesz alap (ground) argumentumok megjelölésére, ami ezeknek a metódusoknak egy még típusbiztosabb használatát teszi majd lehetővé. Az 3. ábra a cél metódusok használatát szemlélteti.

A Prolog4J további innovatív jellemzője, hogy a cél metódusok törzsét automatikusan generálja a Java Compiler API-n keresztül [13]. Ez a képesség csak a Sun Java fordítóját használva működik.

3.1.2 OOLibrary

Az OOLibrary egy tuProlog programkönyvtár, amely egyben a Prolog4J rendszer része. A programkönyvtárnak kettős célja van. Fő célja objektumorientált stílusú programozás lehetővé tétele Prologban. A programkönyvtár egységes típusrendszert definiál. A Prolog atomi típusain túl az összetett termekhez is típust rendel, vannak lista típusok és generikus típusok is. Új típusok is definiálhatók.

Az OOLibrary segítségével osztályhierarchia definiálható. Az osztályoknak lehetnek mezőik, és lehet egy ősosztályuk (egyszeres öröklődés). Az

osztályhierarchia célja az objektum polimorfizmus bevezetése Prologba: bár az osztályok példányai közönséges (összetett) termek, egy osztály példánya szerepelhet a program bármely pontján, ahol az ősosztály objektuma szerepelhet, és emellett a viselkedése saját osztályára jellemző marad.

Metódusokat is definiálhatunk. Ezek nem képezik az osztálydefiníció részét. A formális paramétereik típusa meg van adva a formális paraméterlistán. Az aktuális paraméterek típusa a metódus hívásakor kerül ellenőrzésre.

A programkönyvtár másodlagos célja a Prolog és Java közötti integráció javítása. Ha olyan termet kell Java objektummá konvertálni, amely egy OOLibrary osztálynak példánya, akkor a Prolog4J a megfelelő, `@Term` annotációval ellátott Java osztály példányát fogja létrehozni. Hasonlóan, ezen osztályok objektumai OOLibrary objektumokká konvertálhatók. Ez jelentősen javítja a Java és Prolog nyelvek kapcsolatát, mivel az alkalmazási szakterület ugyanazon osztályhierarchiája használható mindkét nyelvben, és a típuskonverzió közöttük automatikusan megtörténik.

3.1.3 Japlo

A Japlo nyelv [4] a Java-Prolog integráció alternatív módját biztosítja, ami a területen elért első eredményeim egyike. A nyelv a Java nyelv kiterjesztése, amelyben Prolog szabályok Java-szerű szintaxissal adhatók meg. A nyelvtant úgy alkottam meg, hogy olyanok számára is könnyen elsajátítható legyen, akik a Java használatában járatosak, a Prologéban viszont nem.

Számos Java elv használható a szabályok megfogalmazásakor. Például használható néhány vezérlési szerkezet (`if`, `while`, `do-while`), a szabályoknak visszatérési értéke lehet, akárcsak a metódusoknak, és az ilyen szabályok alkalmazása kifejezésben is lehetséges. A másik oldalt tekintve, a szabályok egyszerűen alkalmazhatók Javából: egyrészt az unáris `+` operátor segítségével a megoldás létezése dönthető el, másrészt a szabály alkalmazásával kapott megoldások egy for-each ciklussal roppant egyszerűen bejárhatók.

A Japlo nyelvtana kényelmes módot biztosít Javából elérhető szabályok írására. Megmutattam, hogy egy Japlo szabály sokkal tömörebben használható, mint ugyanez JLogban, ha a szabály alkalmazásának eredménye futás közben előálló értékektől függ. A Japlo másik előnye, hogy statikus típusellenőrzést tesz lehetővé a szabályokra.

3.2 Logikai metaprogramozási keretrendszer

Kifejlesztettem egy logikai metaprogramozási keretrendszert Javához. A keretrendszer alapja a Prolog4J és az OOLibrary. Két rétege van. Az

```
isSingleton(C: classSpec, M: methodSpec, F: fieldSpec) :-
  methodOf(C, M), isPublic(M), isStatic(M),
  returnTypeOf(M, CDesc), descriptorOf(C, CDesc), argTypesOf(M,[]),
  fieldOf(C, F), isPrivate(F), isStatic(F), descriptorOf(F, CDesc),
  reads(M, F),
  methodOf(C, Ctr), isConstructor(Ctr),
  (not(isPrivate(Ctr)) -> !, fail).
```

Figure 4: Az isSingleton metódus

alacsony szintű réteg a program metainformációit tények formájában rögzíti (metaadatbázis). A magas szintű réteg OOLibrary osztályokat és metódusokat definiál az adatbázis tartalmának vizsgálata céljából.

Természetesen új metódusok is szabadon definiálhatók a meglévőek segítségével. Ez lehetővé teszi a program logikai elemzését. A 4. ábra egy egyszerű példát ad erre. Az `isSingleton` metódus eldönti, hogy az „Egyke" tervezési minta [6] alkalmazva van-e egy osztályra vagy sem. Amennyiben az osztály egyke, akkor a mintában részt vevő metódus és mező az `isSingleton` metódus paramétereihez lesz rendelve.

A keresett metainformáció eléréséhez csupán egy lekérdezést kell megadni, amely megadja a kívánt adat tulajdonságait és a közöttük rejlő logikai összefüggéseket. A keretrendszeren belül elérhető a metódusok törzse is OOLibrary objektum formájában, amely így a Prolog eszközeivel átalakítható. Továbbá a metainformáció a bájtkódból lesz kinyerve, nem a forrásból, ami a keretrendszer dinamikusabb felhasználását teszi lehetővé.

3.3 Aspektusorientált programozási keretrendszer

Kifejlesztettem egy aspektusorientált keretrendszert az előbb tárgyalt DMP keretrendszer fölé. Ez lehetővé teszi, hogy a programozó a program végrehajtásának eseményeire hivatkozzon, és így bizonyos alprogramok (ún. *javaslatok)* meghívását ehhez kösse. A program futásának pontjait kijelölő leírások (a vágáspontok) megadása Prolog lekérdezések formájában történik. A program kódját felderítő predikátumok mellett az események térbeli és időbeli viszonyai szinték kifejezhetők. A vágáspontokat annotációk formájában lehet metódusokhoz kötni.

Megmutattam, hogy a vágáspontok kifejezőereje magas, két okból is. Egyrészt, mivel a vágáspontok a program elemeire metaváltozók segítségével hivatkozhatnak, és ezen elemek tulajdonságai és a köztük lévő kapcsolatok a DMP keretrendszer eszközeivel vizsgálhatók. Másrészt azért, mivel az események közötti időbeli viszonyok is kifejezhetők.

A metaváltozók használata növeli a vágáspontok robusztusságát a program evolúciója tekintetében, mivel lehetővé teszi, hogy a vágáspontokat szemantikájuk segítségével írjuk le ahelyett, hogy a kódbeli szintaktikai megjelenésükre hivatkoznánk.

Az AOP rendszerem további előnye, hogy napjaink talán legelterjedtebb programozási nyelvére épül, nem pedig egy kutatási prototípus nyelvre. (Nem elvitatva a prototípus nyelvek fontosságát.) Emellett az aspektusok az osztályok bájtkódjába lesznek beleszőve, nem a forrásukba. Ez lehetővé teszi a betöltési idejű szövést. A Java Debugger Interface (JDI) felületen keresztül futási szövés szintén lehetséges. Egy korábbi cikkemben megmutattam, hogy a futási idejű szövés akár javíthatja is a program teljesítményét a betöltési idejűhöz képest.

3.4 Kontrollált természetes nyelvi felület

Kifejlesztettem egy kontrollált természetes nyelvi felületet az AOP keretrendszer vágáspont definíciós nyelve fölé. A kontrollált természetes nyelvek tökéletesen alkalmasak vágáspontok megadására, számos okból:

- Speciális „szakterületet" képeznek. Itt a szakterület nem a program alkalmazási szakterülete, hanem a metaprogramozás, mint olyan.
- A vágáspontoknak jól definiált, egyszerű szerkezete van, ami két előnnyel is jár:
  - A programozót nem fogja megkötni a kontrollált nyelv merev szerkezete.
  - A nyelv viszonylag könnyen megalkotható. (Ez csupán megvalósítási kérdés.)
- A vágáspontokat deklaratív módon kell megadni ahhoz, hogy azok törékenységét elkerüljük. A deklarációk (kijelentések) a természetes nyelvek legalapvetőbb kommunikációs formái.

Bár a természetes nyelvű szövegek jellemzően bőbeszédűbbek a formális leírásoknál, számos nyelvtani technikát építettem a nyelvbe, amelyekkel a vágáspont-definíciók – amennyire csak lehet – tömören tarthatók. Megmutattam, hogy a kontrollált nyelvi megfogalmazás tömörebb, mint az azonos jelentésű definíciók AspectJ vagy ALPHA nyelveken. Egy speciális példa látható az 3. táblázatban.

Bár a természetes nyelvű szövegek eredendően többértelműek lehetnek, ez (jellemzően) nem igaz a kontrollált természetes nyelvekre. Az általam

Nyelv	*Vágáspont*	*Hossz*
AspectJ	set(FigureElement.x) || set(FigureElement.y) || set(FigureElement.start) || set(FigureElement.end)	98
LogicAJ	(set(?T.x) || set(?T.y) || set(?T.start) || set(?T.end)) && equals(?T, FigureElement)	85
CNL interface	changing x, y, start or end of a 'FigureElement'	48

Table 3: Azonos jelentésű vágáspont-definíciók hossza

készített nyelvtan nem enged meg többértelműséget, így az utalószavak használatát sem.

Egy összetettebb példa látható a 4. táblázatban. Ez a példa a CNL felület néhány egyéb előnyét mutatja be, amelyek sokkal lényegesebbek a tömörségnél. Először is, a CNL vágáspont-definíciók könnyen olvashatók vagy akár megfogalmazhatók olyanok számára is, akik egyáltalán nem jártasak az aspektusorientált programozás terén. Másrészt, elrejtik a végleg elrejtik Prolog belső használatát, ezáltal megkímélve a programozót egy másik programozási nyelv megtanulásától, illetve használatától.

ALPHA	*Kontrollált természetes nyelv*
set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, this.d, 'drawAll', _)), cflow(T1, T2), reachable(Q, P), instanceof(Q, 'FigureElement')	changing F of P, calling drawAll of d last time at T, reading F of P during T, P is reachable from a 'FigureElement'

Table 4: A "cflowreach" vágáspont ALPHA-ban és CNL-ben

4 Összefoglalás

Az értekezés eredményeit összefoglalva kijelenthető, hogy az általam kifejlesztett Java/Prolog többparadigmás programozási keretrendszer (Prolog4J) könnyebben használható mint a jelenleg elérhető hasonló célkitűzésű projektek, több okból is. Egyrészt, mivel a rendszerem kihasználja a Java 5 új elemeit (for-each ciklus, automatikus csomagolás, annotációk). Másrészt a Java objektumok és Prolog termek között automatikus konverziót végez összetett típusok esetén is anélkül, hogy új típusrendszert vezetne be. Végül, mivel lehetővé teszi egyszerű programozói felület kialakítását Prolog lekérdezések

fölé annotált Java metódusok formájában, melyek törzsét a rendszer automatikusan generálja.

Szintén kifejlesztettem egy deklaratív metaprogramozási keretrendszert, amely lehetővé teszi Java programokra vonatkozó állítások megfogalmazását. A DMP rendszer Java osztályok bájtkódjának feldolgozására épül, így ideális alapja lehet egy dinamikus, aspektusorientált programozási rendszernek.

Az általam létrehozott AOP keretrendszernek számos előnye van a jelenlegi rendszerekkel szemben. Egyrészt, támogatja a metaváltozók használatát, ami a vágáspontok megadásának sokkal robusztusabb módját teszi lehetővé. Másrészt a csatlakozási pontok időbeli viszonya is kifejezhető. További előny, hogy az aspektusok szövése dinamikusan, betöltési vagy futási időben történik. Bár ezen képességek külön-külön már megjelentek AOP rendszerekben, egyikük sem támogatja ezek mindegyikét.

Végül, kialakítottam egy kontrollált természetes nyelvi felületet a vágáspontok megadására. Ez egy újszerű megközelítés, amely jelentősen javítja a vágáspontok olvashatóságát, és lehetővé teszi vágáspontok megadását olyanok számára is, akik egyáltalán nem jártasak az aspektusorientált programozás terén. Az ilyen vágáspont-definíciók nem csak könnyen érthetőek, de jellemzően tömörebbek is mint az azonos jelentésű vágáspontok egyéb AOP nyelvekben.

References

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects (Monographs in Computer Science)*. Springer, April 1998.

[2] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

[3] Miklos Espak. Improving Efficiency by Weaving at Run-time. In *In 5th GPCE Young Researchers Workshop 2003*, 2003.

[4] Miklos Espak. Japlo: Rule-based Programming on Java. *Journal of Universal Computer Science*, 12(9):1177–1189, 2006.

[5] Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English (ACE) Language Manual, Version 3.0. Technical Report 99.03, Department of Computer Science, University of Zurich, August 1999.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[7] Object Management Group. Semantics of Business Vocabulary and Business Rules (SBVBR), v1.0. http://www.omg.org/spec/SBVR/1.0/PDF, 01 2008.

[8] Glendon Holst. JLog - Prolog in Java. http://jlogic.sourceforge.net.

[9] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. A Model-Driven Pointcut Language for More Robust Pointcuts. In *Proceedings of the 4th International AOSD Workshop on Software Engineering Properties of Languages for Aspect Technology (SPLAT'06)*, Bonn, Germany, mar 2006.

[10] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[11] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond AOP: toward naturalistic programming. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 198–207, New York, NY, USA, 2003. ACM.

[12] Tom Mens, Roel Wuyts, Kris De Volder, and Kim Mens. Declarative Meta Programming to Support Software Development: Workshop Report. *SIGSOFT Softw. Eng. Notes*, 28(2):1, 2003.

[13] Sun Microsystems. JSR-000199 JavaTM Compiler API. http://jcp.org/aboutJava/communityprocess/final/jsr199/index.html.

[14] European Association of Aerospace Industries (AECMA). AECMA Simplified English (SE) Guide. Available through various AECMA-appointed publishers, (The current version is Issue 1 Revision 2; SE has been regularly updated since 1986.), 2001.

[15] Nestor Rychtyckyj. Standard Language at Ford Motor Company: A Case Study in Controlled Language Development and Deployment. 2006.

[16] Paul Singleton, Fred Dushin, and Jan Wielemaker. JPL: A bidirectional Prolog/Java interface. http://www.swi-prolog.org/packages/jpl/.

[17] Jan Wielemaker. SWI-Prolog 5.4.3 Reference Manual, 2004.

Publications

Article in international journal

[1] Espák M.: Japlo: Rule-based Programming on Java, Journal of Universal Computer Science, vol. 12, no. 9 (2006), 1177-1189

- Cimadamore, M. and Viroli, M.: A Prolog-oriented extension of Java programming based on generics and annotations, Proceedings of the 5th international symposium on Principles and practice of programming in Java, (2007), 197-202, ISBN 978-1-59593-672-1
- Cimadamore, M. and Viroli, M.: P@J: extending Java with declarative programming, 6th International Workshop on Multi-paradigm Programming with Object-Oriented Languages - 31th, July 2007

Refereed articles in international conference publications

[2] Espák M.: Improving Efficiency by Weaving at Run-time, Generative Programming and Component Engineering, Young Researchers Workshop, 2003, Erfurt, Tagungsband, 415–421. ISBN 3-9808628-2-8 http://citeseer.ist.psu.edu/636848.html

- Frei, A. and Grawehr, P. and Alonso, G.: A Dynamic AOP-Engine for .NET, Technical Report 445 of the ETHZ Department of Computer Science
- Frei, A.: Jadabs - An Adaptive Pervasive Middlerware Architecture Dissertation ETH No. 16273, October 2005.

[3] Espák M.: Aspect-Oriented Programming on Lisp, International Conference on Applied Informatics, 2004, Eger, vol. 1, 137-145p. Zbl. 1074.68507

Conference presentations

In English

[4] Espák M.: Improving Efficiency by Weaving at Run-time, GPCE, Young Researchers Workshop, 2003, Erfurt

[5] Espák M.: Innovative Uses of Programming Constructs Supporting Aspect-Oriented Programming, CSCS 2004, The Fourth Conference of PhD Students in Computer Science, 2004, Szeged

[6] Espák M.: Querying on Java Code, International Conference on Applied Informatics, 2007, Eger

In Hungarian

[7] Espák M., Nagy I., Vég Cs.: Cserélhető adatkapcsolat és adattárolás, IV. Országos Objektumorientált Konferencia, 2000, Budapest

[8] Espák M.: Új paradigmák a szoftverfejlesztésben, V. Országos Objektumorientált Konferencia, 2002, Dobogókő

[9] Espák M.: Aspektusorientált programozás megvalósítása metaobjektum-protokollal, Informatika a felsőoktatásban, 2002, Debrecen, http://www.date.hu/if2002

[10] Espák M.: Szabályalapú programozás Javában, Informatika a felsőoktatásban, 2005, Debrecen, http://agrinf.agr.unideb.hu/if2005/kiadvany/papers/E64.pdf

Invited lectures

[11] Espák M.: Többparadigmás programozás, VIII. Béla Gyires IT Lectures, Debrecen, 17th May 2007