Available online at www.sciencedirect.com

SCIENCE DIRECT

ELSEVIER Electronic Notes in Theoretical Computer Science 107 (2004) 33–49

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Integrated Development Environment for Pattern Matching Programming

Julien Guyon, Pierre-Etienne Moreau[1] and Antoine Reilles

LORIA & CNRS & INRIA
Campus Scientifique, BP 239, 54506 Villers-lès-Nancy Cedex France

Abstract

Tom and ApiGen are two complementary tools which simplify the definition and the manipulation of abstract datatypes. Tom is an extension of Java which adds pattern matching facilities independently of the used data-structure. ApiGen is a generator of abstract syntax tree implementations which interacts naturally with Tom. In this paper, we show how Eclipse can be extended to support the development of Tom programs. By integrating a Tom editor, an automatic build process, and an error management mechanism, we demonstrate the integration of an algebraic programming environment in Eclipse. Hence, our work contributes to the promotion of formal methods and Eclipse to the educational, algebraic, and industrial communities.

Keywords: Java, Eclipse, Integrated Development Environment, algebraic programming enviroments.

1 Introduction

As mentioned in [8], the Eclipse Platform is an Integrated Development Environment (IDE) for anything, and nothing in particular. Although the Eclipse Platform has a lot of built-in functionality, it is possible to extend the system. The plug-in is the smallest functional unit of Eclipse Platform that can be developed and delivered separately. All of the Eclipse Platform's functionality is delivered through the interface of plug-in, except for a small kernel known as the Runtime Platform. A simple feature (online help for example) can be added through a single plug-in and developed without any coding at

[1] http://www.loria.fr/~{}moreau

doi:10.1016/j.entcs.2004.02.045

all. A complex one (a complete IDE for a programming language) can be split in several plug-ins, each one coded in Java. So, the tools plugged into the Platform supply the specific features that make it suitable for developing new kinds of applications.

Eclipse is delivered with a tool for Java. The Java development tooling (JDT) adds Java program development capability to the Platform, like any full-featured Java IDE (project management, source code editor, refactoring support, complex searches, incremental compilation, debugging support, *etc.*). Eclipse also supports a C/C++ equivalent IDE: the CDT. All these tools have been developed using the Eclipse framework API, which is an open and documented based framework. The development of a specific tool is very easy. This is exactly what we have done for an algebraic extension of Java: Tom [2].

Object-oriented languages are very popular. However, they still lack powerful features like expressive pattern matching provided by functional programming languages. In practice, these useful features interact poorly with the data abstraction mechanisms which are central to object-oriented languages. Thus, expressing some computations is awkward in object-oriented languages.

In this paper, we present an Eclipse plug-in for Tom [13] and ApiGen [5]. Tom is an extension of Java (and C) which adds pattern matching facilities to these languages [15,16,12,10]. This is particularly well-suited when describing various transformations of structured entities like, for example, trees/terms and XML documents. From an implementation point of view, it is a compiler which accepts different *native languages* (C or Java) and whose compilation process consists in translating the matching constructs into a given underlying native language. Its design follows our experiences on the efficient compilation of rule-based systems [9].

ApiGen is a generator of abstract syntax tree implementations. Taking a concise definition of an abstract datatype, it generates an efficient and strongly typed implementation code for this abstract datatype. The implementation features an efficient memory management and fast equality checking using maximal sharing[2].

As illustrated by Figure 1, ApiGen interacts very well with Tom. In addition to the generated Java code, ApiGen also generates algebraic signatures which can be directly reused in Tom to define pattern matching expressions.

Tom and ApiGen provide support for list representations and list-matching [3] which are very useful for practical applications like XML document transformations. Indeed, XML documents can be easily represented by means of list

[2] available at `http://tom.loria.fr`

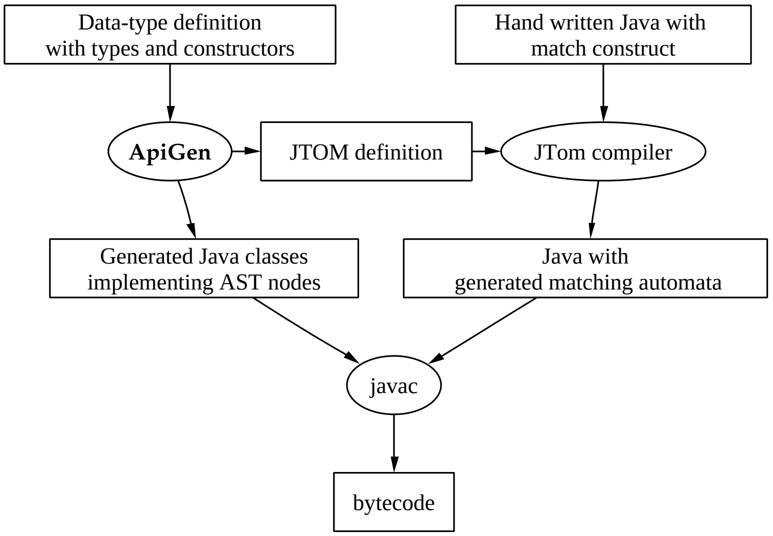

Data-type definition with types and constructors
Hand written Java with match construct
ApiGen
JTOM definition
JTom compiler
Generated Java classes implementing AST nodes
Java with generated matching automata
javac
bytecode

Fig. 1. ApiGen generates a datatype in Java and a definition of this datatype as input for TOM. The users writes code using the match construct on the generated Abstract Syntax Tree classes, and TOM compiles this to normal Java.

operators.

The rest of this paper is structured as follows. Section 2 briefly presents TOM and ApiGen. The Eclipse plug-in and its features are described in Section 3. Section 4 presents the two main contributions of this work. On one hand, it promotes Eclipse and provides a complete IDE to the algebraic programming community. On the other hand, it shows that Eclipse is generic enough to support the development of new "non-standard" programming environments. Section 5 present some related work. The last section concludes the paper and presents perspectives to this work.

2 Tom and ApiGen

2.1 *Pattern matching with* TOM

For sake of simplicity, we only consider two additional new constructs of TOM extending the Java syntax: `%match` and *back-quote* (‘). The first construct is similar to the `match` primitive of ML and related languages: given a term (called subject) and a list of pairs pattern-action, the `match` primitive selects a pattern that matches the subject and performs the associated action. This construct may thus be seen as an extension of the classical `switch/case` construct. The second construct is a mechanism that allows one to easily build ground terms over a defined signature. This operator, called *back-quote*, is followed by a well-formed term written in prefix notation.

In order to give a better understanding of TOM's features, let us consider a simple symbolic predicate (`greaterThan`) defined on Peano integers built using the *zero* and *successor* symbols. The comparison of two integers can be described in the following way:

```
boolean greaterThan(Nat t1, Nat t2) {
  %match(Nat t1, Nat t2) {
    x,x           -> { return true; }
    suc(_),zero() -> { return true; }
    zero(),suc(_) -> { return false; }
    suc(x),suc(y) -> { return greaterThan(x,y); }
  }
}
```

This example should be read as follows: given two terms `t1` and `t2` (that represent Peano integers), the evaluation of `greaterThan` returns `true` if `t1` is greater or equal to `t2`. This is implemented by (non linear) pattern matching (first pattern) and anonymous variables (second and third patterns). The reader should note that variables do not need to be declared: their type is automatically inferred from the definitions of the operators in which they are involved. To distinguish a constant from a variable (*e.g.* the constant `zero`), empty braces could be used (`zero()`).

As mentioned previously, an important feature of TOM is to support list matching, also known as associative matching with neutral element (inspired by the ASF+SDF Meta-Environment [3]). Let us consider the associative operator `conc` used for building list of naturals (`NatList`). In TOM, an associative operator is a variadic operator and each sub-term could be either of sort *element* or *list* (respectively `Nat` or `NatList` in our case). To illustrate the expressivity of associative matching, let us define a sorting algorithm:

```
public NatList sort(NatList l) {
  %match(NatList l) {
    conc(X1*,x,X2*,y,X3*) -> {
      if(greaterThan(x,y)) { return `sort(conc(X1*,y,X2*,x,X3*)); }
    }
    _ -> { return l; }

  }
}
```

This example illustrates the use of *list variables*, annotated by the '`*`': such a variable can be instantiated by a (possibly empty) list. Given a partially sorted list, the `sort` function tries to find two elements `x` and `y` such that `x` is greater than `y`. If two such elements exist, they are swapped and the `sort` function is recursively applied. When the list is sorted the first pattern cannot be found in the list and the next pattern is tried. In fact, this second pattern imposes no restrictions on its subject and thus the corresponding action is triggered and the sorted list `l` is returned.

2.2 Generation of abstract datatypes

As mentioned in Section 1, ApiGen takes a concise definition of abstract datatype (an `*.adt` file) and generates an efficient and strongly typed Java implementation. Considering the previous Peano based example, the datatype definition is given in Figure 2. A particularity of Tom is to be data-structure independent. Thus, given a concrete Java implementation, a Tom signature has to be defined to establish a mapping between implementation and algebraic constructors. In addition to the generated Java code, ApiGen also generates this Tom signature definition.

```
datatype Peano
  Nat     ::= zero
            | suc(pred:Nat)
  NatList ::= conc(elt:Nat)
```

Fig. 2. A datatype definition for Peano integers and list of naturals.

An interesting point offered by abstract datatype is to be statically typed: each constructor is defined by a domain and a co-domain. In practice, this reduces the risk of programming errors since an ill-formed term (or pattern) is detected at compile time.

2.3 XML transformations

Another application of Tom is to support the manipulation of XML documents. Indeed, an XML document can be parsed and represented by a tree-based data-structure like Document Object Model (Dom), promoted by the W3C. Dom is a platform independent interface that allows to dynamically access and update the content, structure, and style of XML documents. Tom provides a mapping mechanism (the signature definition) that enables us to see a Dom tree as an algebraic term, on which pattern matching can be performed. In addition to the standard prefix notation, Tom supports a specific XML-like notation which makes pattern definitions very natural for XML users. As an example, comparing two XML nodes (`Person`) according to values of their attributes (`Age` in this example) could be described as follows:

```
public int compare(TNode t1, TNode t2) {
  %match(TNode t1, TNode t2) {
    <Person Age=a1></Person>,
    <Person Age=a2></Person>
    -> { return a1.compareTo(a2); }
  }
  return 0;
}
```

When using the XML notation, the pattern `<Person>...</Person>` matches an XML document only when this document is rooted by a `Person` node. In our example, the notation `Age=a1` means that we are only interested in a node which contains the attribute `Age`. For these nodes, the value of the attribute (a string) is stored in the fresh variable `a1`. The previous pattern matches when given two XML documents `t1` and `t2`, these documents are rooted by `Person` and have an attribute `Age`. In this case, the value of their attribute `Age` are stored respectively in `a1` and `a2`. Thus, we say that `t1` is smaller than `t2` when `a1` is smaller than `a2` (using the `compareTo` method provided by the `String` library of Java). In all other cases, the two documents `t1` and `t2` are not comparable and the integer `0` is returned.

With this XML notation, it becomes possible to retrieve information stored in subterms. The following pattern matches an XML document when it contains a sub-node rooted by `Name`. In this case, the content of the `Name` node (a string) is stored the variable `name`.

```
<Person Age=a1><Name>#TEXT(name)</Name></Person> -> ...
```

Object-oriented languages are not good at analyzing and transforming XML, as one can observe when manipulating XML documents. This is because such trees usually contain data but no methods. The DOM library helps the programmer but make transformations still tedious to express. The experience showed us that TOM performs well in this area, mainly because the notion of pattern matching naturally extends to the processing of XML data. Ideally, one would hope for a fusion which unifies concepts found in different paradigms. Our proposal is a first answer to this problem. As an example, the DOM library offers a way to obtain all nodes with a specific name. This can be specified in TOM, but it is more tedious. The collection of all nodes with a particular attribute is more easy in TOM and very tedious using the DOM library.

The complementary approaches of TOM and DOM provides a very powerful paradigm which offers a unified, intuitive, and simple syntax to describe XML analysis and transformations. This approach competes with other technology like XSLT, but it is safer since we benefit from the static typing of TOM and Java.

3 Integration of Tom into Eclipse

Promoting TOM implies to develop all expected tools to exploit such kind of new technology. Based on our experiences on the efficient compilation of

ELAN [9], a strong core compiler for the language has been developed [13]. This compiler, called JTOM, is written in Java and TOM itself, using ApiGen for abstract datatype definitions. In parallel, a dynamic debugger, able to track pattern matching and rewrite step expressions has been implemented. This tool allows to set conditional break-points and visualize variables instantiated by pattern matching.

Recently, the TOM compiler has been adapted to be integrated into the Eclipse platform. The development of the TOM plug-in, thanks to Eclipse framework simplicity, led to the emergence of a complete integrated algebraic environment dedicated to TOM. This work share some common features with the ASF+SDF Meta-Environment [1,4].

3.1 Migrating the system to Eclipse: expected gains

Originally, JTOM and ApiGen were implemented in Java as command line tools. Traditional editors, like Emacs and Vim, as well as Make commands were used for editing and compiling the source files. This approach is suitable for experienced Unix users, however it makes the installation of the system more complex for other users, because editors, scripts, and shell variables have to be tailored. Although, it is possible to distribute everything in a single package, it still requires an external editor and some script customizations.

There was definitely a need for a more friendly environment to promote the language and also make our development easier. The Eclipse platform answers to these problems by providing an integrated environment which contains both editors, compilers, and builders. Furthermore, Eclipse is able to provide a complete specialized editor for TOM, offering syntax coloring, keyword completion, and error reporting in a uniform environment.

Finally, Eclipse also provides debugging support which could be reused to integrate the current TOM (textual mode) debugger.

3.2 Problems to solve

After studying the different opportunities offered by the Eclipse platform, we have identified several problems to solve:

- how to keep the TOM system a self-contained package (independent to Eclipse) as this migration should not break the command line tool capabilities ?
- how to provide a editor with TOM syntax coloring capabilities?
- How to re-use the JDT editor for editing and refactoring parts of TOM programs which are written in Java?

- how to extend the JDT Java editor and keep the Tom plug-in as independent as possible of JDT's low-level implementation details ? This is important to avoid extra work each time the Eclipse platform is upgraded.
- how to integrate JTom and ApiGen in such a way that their compilation becomes automatic and transparent to the user ?
- how to coordinate the different compilers (javac, JTom, and ApiGen) to smoothly integrate the errors raised by these tools ?
- how to manage the execution of the external tool into the same Java virtual machine as Eclipse without unexpected behavior ? For example, the tools should terminate properly without stopping the virtual machine.

Some of these problems can be solved by adapting our tools to fit in the proposed framework: removing static reference and avoiding `System.exit` for example. However, some others issues, related to the JDT editor for example, are more difficult to solve and can only be lighten by adopting a "non-intrusive" approach or waiting for a more flexible design of the Eclipse environment.

3.3 The Tom *plug-in*

The presented plug-in is fully functional and is available at Tom web page. The system consists of about ten *Eclipse extension points* [8] to deliver services ranging from wizards to complex structured editor and automatic incremental build support.

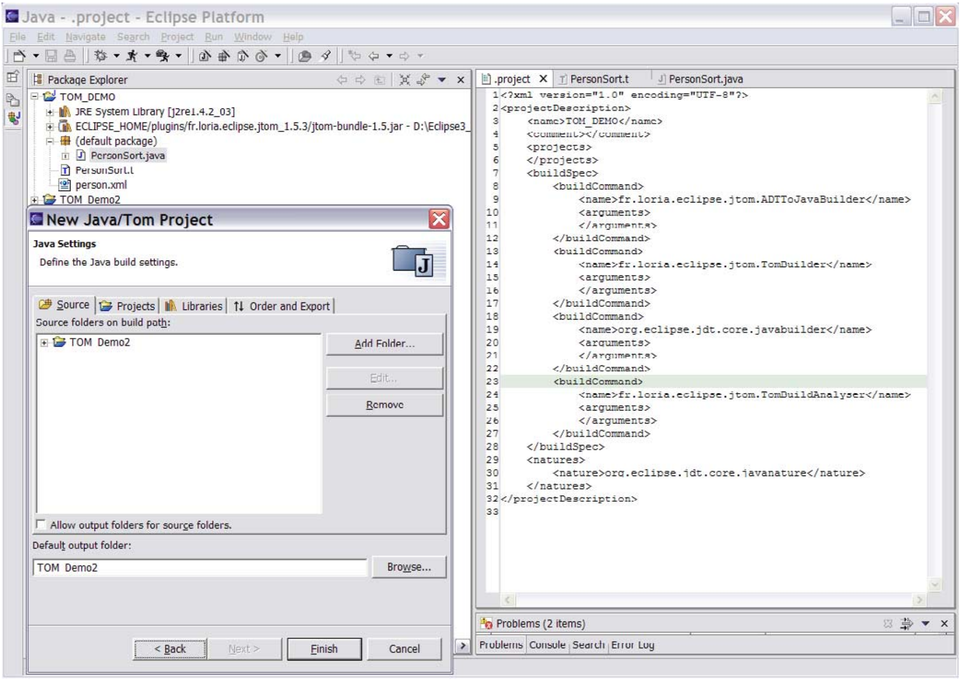
Project and resource wizards

A wizard aims at simplifying the creation of new projects. Similarly to Java, the JTom wizard allows the programmer to define the inheritance hierarchy of classes, as well as the skeleton, based on Java templates.

In addition, as illustrated in Figure 4, the JTom and ApiGen wizards also invite the programmer to define specific properties, like compilation options for example.

Structured editor

To each Tom file is associated a specific editor, whose design is inspired by the Java editor. On the implementation side, some behavior is inherited from the classes included in the JDT. In particular, this gives Java coloring, indentation, and function template completion for free. The specific part consists in adding Tom functionality to complete the inherited one:

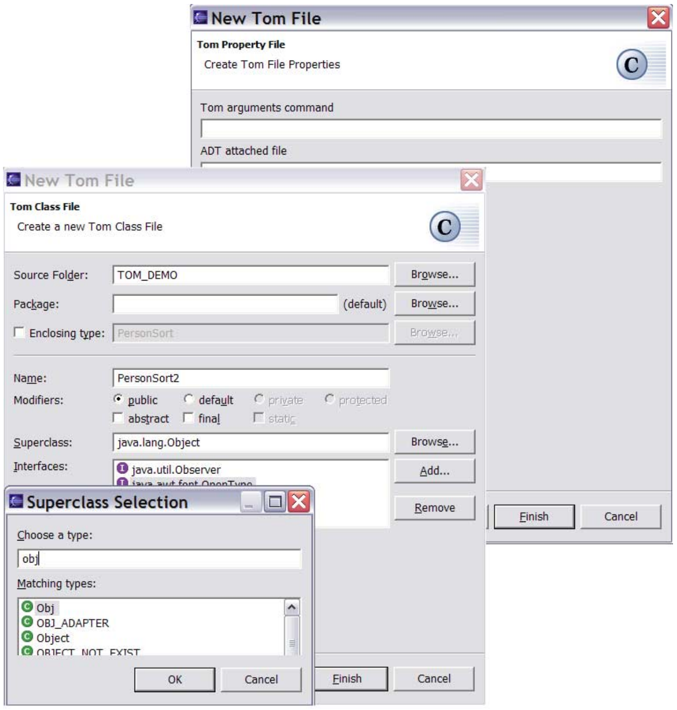


```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
    <name>TOM_DEMO</name>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>fr.loria.eclipse.jtom.ADTToJavaBuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>fr.loria.eclipse.jtom.TomBuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments>
            </arguments>
        </buildCommand>
        <buildCommand>
            <name>fr.loria.eclipse.jtom.TomBuildAnalyser</name>
            <arguments>
            </arguments>
        </buildCommand>
    </buildSpec>
    <natures>
        <nature>org.eclipse.jdt.core.javanature</nature>
    </natures>
</projectDescription>
```

Fig. 3. The Tom project wizard extends the Java one and configures the project to make automatic the compilation of Tom programs

- syntax coloring has been extended to consider and highlight Tom constructs,
- the auto-completion word processor has been extended to perform Tom keyword completion, depending on the context,
- a last feature allows to display the signature of an algebraic constructor. By double-clicking on a Tom constructor, an info-pop gives its algebraic signature (domain, co-domain, and field-names). This considerably improves the development process and participates to reduce the number of ill-formed term errors.

The last point is interesting because it illustrates the smooth integration of Tom and ApiGen: algebraic constructors are used in Tom programs, whereas their signature is defined at the ApiGen level (in `*.adt` files). By parameterizing the Tom editor with signature definitions, the editor is able to retrieve and display pertinent information.



New Tom File
Tom Property File
Create Tom File Properties
Tom arguments command:
ADT attached file:

New Tom File
Tom Class File
Create a new Tom Class File
Source Folder: TOM_DEMO
Package: (default)
Enclosing type: PersonSort
Name: PersonSort2
Modifiers: [✓ public] default private protected
abstract final static
Superclass: java.lang.Object
Interfaces: java.util.Observer, java.awt.font.OpenType

Superclass Selection
Choose a type: obj
Matching types:
Obj
OBJ_ADAPTER
Object
OBJECT_NOT_EXIST

Fig. 4. The TOM wizards allow to specify the expected class hierarchy and other resource properties like compilation options

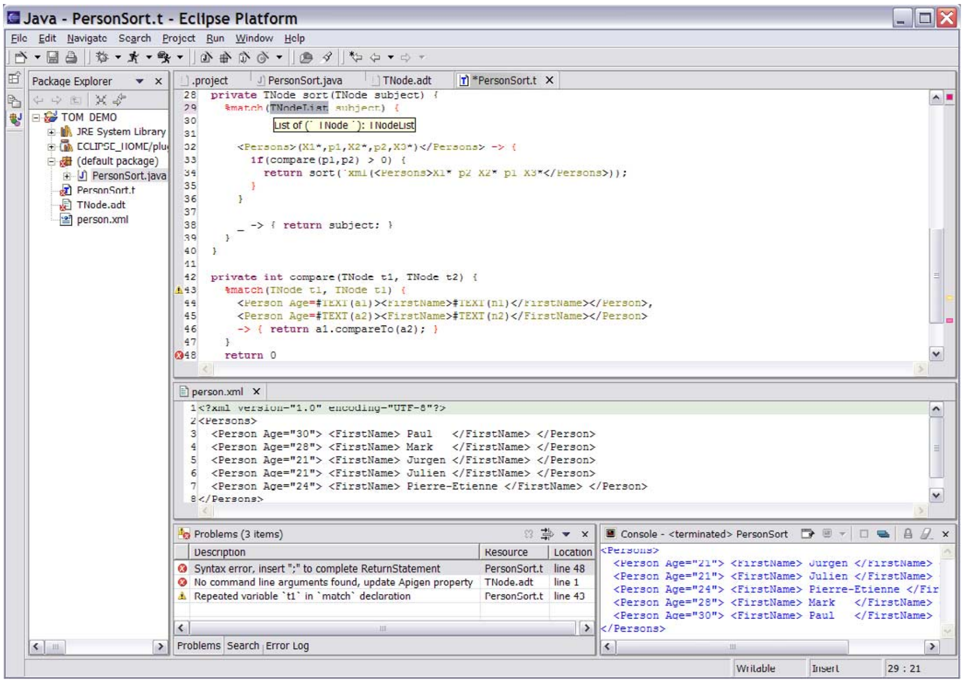
Automatic build process

The Eclipse platform defines an automatic build process, activated each time a resource is modified. The compilation workflow (Figure 1 on page 3) shows how this process can be used to automatically compile a TOM project: each time a resource is modified (an ApiGen or a TOM file), it has to be re-compiled. Moreover, when a signature definition is modified, the depending TOM files also have to be re-compiled. To complete the build process, all generated Java files have to be compiled. This is performed by the third phase of the build process definition (see Figure 3 on the page before).

Error management

When using an integrated development environment, one of the most important desired feature is the possibility to visualize programming errors. This is, in our opinion, a very strong point provided by the Eclipse IDE.

In order to minimize the difference between the TOM environment and the standard Java environment, it was essential to offer the possibility to visualize Java and TOM errors in a uniform way. As illustrated in Figure 5, we managed



```
28  private TNode sort(TNode subject) {
29    %match(TNodeList subject) {
30
31
32      <Persons>(X1*,p1,X2*,p2,X3*)</Persons> -> {
33        if(compare(p1,p2) > 0) {
34          return sort(`xml(<Persons>X1* p2 X2* p1 X3*</Persons>));
35        }
36      }
37
38      _ -> { return subject; }
39    }
40  }
41
42  private int compare(TNode t1, TNode t2) {
43    %match(TNode t1, TNode t1) {
44      <Person Age=#TEXT(a1)><FirstName>#TEXT(n1)</FirstName></Person>,
45      <Person Age=#TEXT(a2)><FirstName>#TEXT(n2)</FirstName></Person>
46      -> { return a1.compareTo(a2); }
47    }
48    return 0
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Persons>
  <Person Age="30"> <FirstName> Paul   </FirstName> </Person>
  <Person Age="28"> <FirstName> Mark   </FirstName> </Person>
  <Person Age="21"> <FirstName> Jurgen </FirstName> </Person>
  <Person Age="21"> <FirstName> Julien </FirstName> </Person>
  <Person Age="24"> <FirstName> Pierre-Etienne </FirstName> </Person>
</Persons>
```

Problems (3 items)

	Description	Resource	Location
⊗	Syntax error, insert ";" to complete ReturnStatement	PersonSort.t	line 48
⊗	No command line arguments found, update Apigen property	TNode.adt	line 1
⚠	Repeated variable `t1` in `match` declaration	PersonSort.t	line 43

Console - <terminated> PersonSort

```
<Persons>
  <Person Age="21"> <FirstName> Jurgen </FirstName>
  <Person Age="21"> <FirstName> Julien </FirstName>
  <Person Age="24"> <FirstName> Pierre-Etienne </Fir
  <Person Age="28"> <FirstName> Mark   </FirstName>
  <Person Age="30"> <FirstName> Paul   </FirstName>
</Persons>
```

Fig. 5. Eclipse workspace in action: Tom and Java errors are visible in the "Package Explorer", the "Problems" window as well as in the editor. The result of the execution (a sorted XML document) is shown in the "Console".

to retrieve and display Tom and ApiGen errors (see the "warning line 43" for example). In some sense, it may be considered as normal since we have a full control over the compiler. The complex part was to be able to retrieve and display all Java errors. When designing a compiler or a pre-processor, it generally introduces new lines and constructs. Thus, it is difficult to maintain a correspondence between error locations in the generated code and their origin in the source code. As an example, how to be sure that an error line n (in a Tom program), will produce an error at the same line in the generated Java code.

Our compilation scheme enables us to introduce a synchronization mechanism which ensures that any *native language* block (C or Java) is kept unmodified (same line and column) in the generated file. Thus, any Java error detected in the generated file can be collected and associated to the source Tom file (see the "error line 48" in Figure 5).

Learned lessons

By providing an automatic build process and a good error management mechanism, the presented Eclipse plug-in considerably simplifies the development of Tom based programs.

By developing such a plug-in, we have identified several opportunities for improvement in both the Tom project as well as in Eclipse. In addition to the design of the source code synchronization mechanism, we had to adapt our tools to allow them to share a common Java virtual machine. Thus, it appeared essential to remove all static objects as well as to banish all calls to `System.exit()`, since it immediately terminates the execution of the Eclipse environment.

On the Eclipse side, we observed that the development of wizards and automatic build processes is straightforward. One of the major issues has been the realization of the editor, in the sense that it was nearly impossible to reuse the JDT code to build the Tom editor. Especially, the parsing phase was difficult to realize within the Eclipse infrastructure. Although Java programs can be "easily" split between single-line comment, multi-line comment, javadoc, and the rest, it is not the case for Tom programs.

A last difficulty we encountered was the ability to stay as independent as possible from the low-level JDT implementation. However, this appears to be essential to avoid extra work, like cross-platform tests, each time there is a new release of JDT or the entire system. In particular, it was not trivial to migrate from the current stable version of Eclipse (2.1) to the new pre-built versions (3.x).

4 Contributions

4.1 A platform for academic community

In the previous section we showed how a pleasant and easy-to-use development environment can be provided for an extension of Java. Moreover, we also showed how the Eclipse framework can be turned into a platform dedicated to refactoring and debugging of algebraic based languages. The presented Tom plug-in makes the build process automatic, compilation errors are reported in the editor and located, with explanatory messages. In our opinion, this contributes to reduce the time spent in debugging and thus to speed-up the development process.

Providing such an environment is with no doubt interesting for the development of formal methods, because it will help students to learn and use algebraic specification languages. The facilities offered by such an environment allow to focus on the underlying concepts of such a language instead of the syntactic idiosyncrasies of the language. By quickly identifying syntax and type errors, the IDE contributes to focus on higher level concepts rather than technical problems.

An integrated development environment for an algebraic language is definitely an interesting platform for making tool presentations, showing examples, modifying, and running them. The professional and easy-to-use interface, may convince industrial users to switch to such a tool and start using more formal techniques in their sofware development process. It is also a convenient way of distributing the system by means of a platform independent plug-in, with a very simple "3-clicks" installation procedure.

Furthermore, for all these reasons, the Eclipse plug-in is also a very convenient tool for experienced algebraic style programmers. In this way, the presented results are a concrete contribution and will certainly promote the emergence of Eclipse to algebraic programming community.

4.2 A validation of Eclipse technology extensibility

This work shows how Eclipse can be used to develop new environments for languages or extensions, using the open architecture of the IDE framework. We have shown how this framework can be used to integrate different tools, formerly considered as independent tools, involving a complex build process into a programmer friendly environment. The introduction of other tools allows to unify these tools in a common platform. We show that the potential of Eclipse environment can be also easily extended to promote such new languages by providing a uniform framework with features usually found in professional used IDE only.

This Tom plug-in is a first step in introducing Eclipse as a platform of greatest interest for the development and the distribution of development environments for algebraic languages like rule-based languages. This shows the interest of a project like the Eclipse project for the academic community, and places Eclipse as a generic support tool for promoting new ideas.

Eclipse can be turned into a generic framework for providing tools and environments for the development of high-level executable algebraic specifications.

Another contribution of this work is to confirm that the Eclipse Platform design is good and generic enough to support the development of new "non-standard" programming environments.

Another encouraging and promising contribution is to highlight some points that have to be improved, like relaxing the deep-connection between the Java grammar and the Eclipse editor for example.

4.3 The plug-in in action

The plug-in has applications in both research and industrial settings. The most important example is the development of JTom in Java and Tom. It represents about 10,000 lines of Tom code converted in more than 30,000 lines of Java. There are about 200 constructors defined with ApiGen generating about 30,000 lines of Java code, with maximal sharing and fast equality check. The project represent almost 80,000 lines of code. Only 6,000 lines of code have been necessary to realize the plug-in. This point validates the extensibility of Eclipse plug-in architecture.

The plug-in has also been used to design and implement various proving and model-checking tools. Recently, a propositional prover has been prototyped: given a proposition to prove, the system generates one or all possible proofs in a readable way (using LaTeX as a typesetting system). In another context, the Needham-Schroeder Public-Key protocol (establishing a mutual authentication between an initiator and a responder) has been described and verified using Tom [7]. In both cases, the interaction between Java and Tom was very fruitful: Tom was used to specify the transition system, whereas the expressive power of Java was used to specify how the search space has to be explored. Finally, the use of Eclipse made this fusion of paradigms so natural that Tom appears to be mature enough to be promoted in a teaching or an industrial environment.

The Tom plug-in has been recently used by the CRIL Technology Group. The main theoretical objective of this research project is to specify and perform transformations of timed automata. From an implementation point of view, most of the data-structure are represented by XML documents. Thus, the need of a powerful and integrated XML transformation tool was essential. A first prototype has been implemented in Java using Eclipse and the Dom library to describe the various transformations. In a second version of the implementation, it was decided to use Tom to describe the transformations. It is worth to say that the existence of the Tom-Eclipse plug-in was essential: without such a plug-in it would have been much more difficult to convince this group to use higher-level programming concepts to implement its tools. The result are quite impressive up-to now, the Eclipse plug-in described in this paper has enabled them to implement complex algorithms, they could never have implemented using Dom. The Tom plug-in also helped them to improve the maintenance of their system and reduce the size of the project by a factor 3 (from 1,200 lines of Java to 400 lines of Tom).

5 Related work

Several systems have been developed in order to integrate pattern matching and transformation facilities into imperative languages. For instance App [14] and Prop [11] are two extensions of C++: the first one is a preprocessor which adds a match construct to the language, whereas the second one is a multi-paradigm extension of C++, including pattern matching constructs. Finally, Pizza [15] is a Java extension that supports parametric polymorphism, first-class functions, class cases and pattern matching. All these approaches are interesting and powerful but less generic than Tom since they strongly depend on the underlying language.

In spirit, Prop and Pizza are very close to Tom: they add pattern matching facilities to a classical imperative language, but the method of achieving this is completely different. Indeed, they are more intrusive than Tom since they really extend C++ and Java with several new pattern matching constructions.

The non-intrusive approach of Tom allows us to be more reactive to any modification of the underlying language (Tom is already ready for Java 1.5 for example) and also simplifies its integration into programming environments like Eclipse.

As far as we know, several extensions of Java have already been integrated into Eclipse. Let us mention the JavaCC and the Jack [6] plug-ins for instance. JavaCC is a parser generator for Java, whereas Jack is a tool for static verification of Java applets, using JML annotations. However, to the best of our knowledge, these tools are not fully integrated into Eclipse, as they do not provide any good support to retrieve and display programming errors in the Eclipse editor.

6 Conclusion and further work

In this paper, we have shown how the Eclipse components could be re-used to deliver a plug-in providing an integrated environment (structured editors, build processes and error management) for an extension of Java: Tom.

In particular, we have discussed how several compilers can be coordinated and adapted to Eclipse in order to display programming errors. However, the JDT editor does not provide extension points which are flexible enough to be re-used in a context independent of the core implementation.

We plan to improve our plug-in with the following new features:

- auto-completion of field-names and algebraic constructors. For this, we need to improve the JTom compiler in such a way that it will support incremental compilation of Tom programs.

- code refactoring capabilities as illustrated by the JDT environment. This point depends on the implementation and the evolutions of the JDT development.
- creating a specialized Eclipse view acting as a graphical interface on top of the existing (textual mode) debugger.

Some of these features raise new problems, involving design modifications of the JTom compiler.

At last, it should be interesting to port all these efforts to the C language by the intermediate of the CDT tools. This port will benefit from the use of conventions and generic functionality imposed by the platform. So, this task should consists in refactoring and adapting the editor and the integration of compilers.

The experiment of integrating Tom and ApiGen in the Eclipse platform showed us the ability of Eclipse to support "non-standard" languages. Eclipse appears to be a platform of choice for new language support in the academic community and for teaching such new languages.

References

[1] M. van den Brand and al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proceedings of Compiler Construction, 10th International Conference*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[2] M. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.

[3] M. van den Brand, J. Heering, P. Klint, and P. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[4] M. van den Brand, H. Jong, P. Klint, and A. Kooiker. A language development environment for eclipse. In M. Burke, editor, *first eclipse Technology eXchange (eTX)*, pages 61–66, 2003.

[5] M. van den Brand, P.-E. Moreau, and J. Vinju. A generator of efficient strongly typed abstract syntax trees in java. Technical report SEN-E0306, ISSN 1386-369X, CWI, Amsterdam (Holland), November 2003.

[6] L. Burdy, J.-L. Lanet, and A. Requet. Java applet correctness kit: http://www.gemplus.com/smart/r_d/trends/jack.html.

[7] H. Cirstea, P.-E. Moreau, and A. Reilles. Rule based programming in Java for protocol verification. In N. Marti-Oliet, editor, *Proceedings of the fifth International Workshop on Rewriting Logic and Applications*, ENTCS. Elsevier Sciences, 2004. To appear.

[8] Eclipse platform technical overview. Technical report, Object Technology International, Inc., 2001.

[9] H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[10] K. Lee, A. LaMarca, and C. Chambers. Hydroj: object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 205–223. ACM Press, 2003.

[11] L. Leung. Prop: `http://cs1.cs.nyu.edu/phd_students/leunga/prop.html`.

[12] J. Liu and A. C. Myers. Jmatch: Iterable abstract pattern matching for java. In V. Dahl and P. Wadler, editors, *Proceedings of PADL'03*, volume 2562 of *LNCS*, pages 110–127. Springer-Verlag, 2003.

[13] P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

[14] G. Nelan. App: `http://www.primenet.com/~georgen/app.html`.

[15] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, USA, 1997.

[16] M. Zenger and M. Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the 6th ACM SIGPLAN International Conference on functional Programming (ICFP'2001), Florence, Italy*, pages 241–252. ACM Press, 2001.