Recovering and checking software architectural properties based on execution tree analysis

Thiago H. Braga[1], **Marcelo de A. Maia**[2], **Roberto da S. Bigonha**[1]

[1]Departamento de Ciência da Computação– Universidade Federal de Minas Gerais (UFMG)
Campus da Pampulha – 31270-010 – Belo Horizonte – MG – Brazil

[2]Faculdade de Computação – Universidade Federal de Uberlândia (UFU)
Campus Santa Mônica – Bloco B – 38400-902 – Uberlândia – MG – Brazil

{thiagohb,bigonha}@dcc.ufmg.br, marcmaia@facom.ufu.br

Abstract. *The specification of the software architecture is an important artifact produced during software design. However, traditional specification techniques do not provide support for guaranteeing that architectural properties of the specification are valid for the implemented software. We propose an approach to recover and specify architectural properties based on execution constraints. The proposed approach is based on specifying constraints as queries in execution trees of software systems. The constraints are generated as annotations and verified against the implementation with a dynamic verifier. We have implemented a tool for Java systems and we argue that our approach presents some benefits over similar ones.*

1. Introduction

The lack of intermediate abstractions that link user requirement features to the concrete implemented features has always been an important concern for software developers. The structural design is an important intermediate level of abstraction that can be represented by the software architecture, which role consists of describing: 1) how the whole system has been partitioned, 2) how those parts relate with each other, 3) how those parts can be cooperatively developed, and 4) how those parts can be grouped to solve a whole problem [Clements et al. 2002].

Some problems can arise in software development regarding its architecture. In some cases, the software is implemented without any predefined architecture. The implementation can be accelerated, but for a long-term project, its maintenance will be impaired because understanding design decisions looking only at the source code will be harder than it would be with a documented system. There are other cases, where the software architecture has been specified, but it may be inconsistent with the current implementation because of source code changes. In this case, an important question is how to identify if the architectural specification is outdated. In other words, the developer needs to guarantee that the system implementation is consistent regarding its architecture specification [Yan et al. 2004]. Application frameworks are almost-done applications for specific domains, that generally define several hot-spots that can be specialized according to domain-specific problems. They enable rapid construction of whole systems. But an important issue that remains is to verify if an instantiation obeys constraints imposed by the framework.

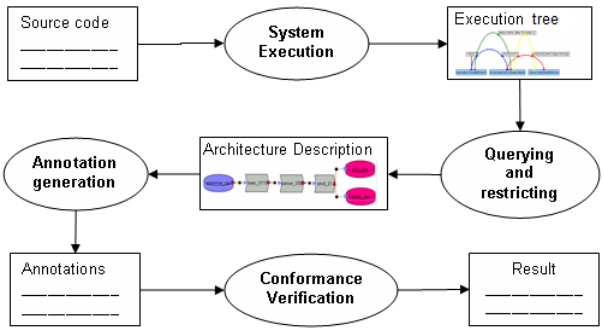

Figure 1. Overview of the proposal

These problems are significant because an implementation that does not obey the architecture specification nullifies the benefits of architectural design.

This work aims at defining mechanisms that enables the specification of constraints on the interaction between components of the system architecture. These constraints can be identified and specified during the process of analyzing the execution tree generated from a scenario execution of the system. The imposed constraints on the system architecture should be verified against its implementation whenever the source code changes. If the changes affect the constraints, the developer is warned. Then, it is up to him to decide either to fix the changes, or to change the architectural to conform to the new version.

This paper is organized as follows. Section 2 presents our solution to recovering and checking architectural properties. In Section 3, we define a methodology for using our approach and apply it in two systems. Section 4 presents the discussion and related work. Finally, in Section 5 the concluding remarks are presented.

2. The proposed solution

This section presents the elements of our approach. The approach is based on the idea of defining queries and constraints in the execution tree of the target system. We have designed and implemented custom languages to specify these queries and constraints that will be shown in Sections 2.2 and 2.3, respectively.

The queries in the execution tree are used to help the developer finding the main components of the software architecture, as well their relationships with each other. The queries enable the developer to observe the creation of objects and their method call interactions.

The constraints in the execution tree are specified with a language similar to the language used to specify queries. They are transformed into annotations that are incorporated in the source code and verified after each program execution by a dynamic verifier implemented as an AspectJ aspect. Figure 1 depicts our approach.

2.1. Execution trees of Java programs

We define a model to represent the execution of Java programs, called execution tree, using some join points definitions of AspectJ [Kiczales et al. 2001]. We are interested in the following execution points: 1) all operations that occur inside a block of a method; 2)

all operations that occur inside a block of a constructor; 3) all operations that occur inside the static initialization block of a class. We will use the term *execution points* considering only these three possibilities.

The scope of the above three execution points defines the *execution context* that can be either: a class context related to the static initialization blocks and to the static methods of a class, or, an object context related to the constructors and non-static methods of a class. The context is used to identify instances of possible components of the system architecture and the execution points are used to represent interactions between these components. Contexts are important to differ several instances of the same class, because each one can have a different role or can perform a different function during the system execution.

The execution tree of a Java program is a tree $G(V, E)$ such that:

- $v \in V \Leftrightarrow v = (c, p)$:
  - c is a context for the execution of point p;
  - p is an execution point.

  If the execution point is non-static then the context is a class instance. If the execution point is static, the context will be represented by an object of the type `Class` corresponding to the class of the current instance.
- for each started `Thread` t, there is a node $v_t = (t,$ `java.lang.Thread.run()`$)$;
- there is an edge $e \in E$ from $v_0 = (c_0, p_0)$ to $v_1 = (c_1, p_1)$ if during the execution of p_0 in the context c_0, the point p_1 has been executed in the context c_1.

The execution tree creator was implemented in AspectJ and is an aspect that is weaved with the target system intercepting the three types of execution points considered in this approach.

2.2. Query language for the execution tree

Execution trees have a large number of nodes, even for the execution of small programs. The main goal of querying execution trees is enabling users to specify queries that filter information about the system execution and to help them identifying what are the main components of the software architecture and their interrelationships. The language allows two types of queries:

- Query of consecutive execution points: after the execution of $\textit{ExecutionPattern}_r$ in $\textit{ContextPattern}_r$, the execution point $\textit{ExecutionPattern}_c$ is immediately executed in the context $\textit{ContextPattern}_c$. The user writes:

  `(`*ContextPattern*$_r$`,` *ExecutionPattern*$_r$`)` **`-->`** `(`*ContextPattern*$_c$`,` *ExecutionPattern*$_c$`)`
- Query of adjacent execution points: if there is a path in any execution subtree of the program, where the point $\textit{ExecutionPattern}_r$ in the context $\textit{ContextPattern}_r$ is the root and the point $\textit{ExecutionPattern}_c$ in context $\textit{ContextPattern}_c$ is a reachable node. The user writes:

  `(`*ContextPattern*$_r$`,` *ExecutionPattern*$_r$`)` **`-+>`** `(`*ContextPattern*$_c$`,` *ExecutionPattern*$_c$`)`

  The initial rules of the query language are:

$$
\begin{array}{rcl}
S & \rightarrow & V_r \texttt{ --> } V_c \\
 & | & V_r \texttt{ -+> } V_c \\
V & \rightarrow & \texttt{(}\ \textit{ContextPattern}\ \texttt{,}\ \textit{ExecutionPattern}\ \texttt{)}
\end{array}
$$

ContextPattern	→	*TypePatternExpr*
TypePatternExpr	→	*OrTypePatternExpr*
	|	*TypePatternExpr* `&&` *OrTypePatternExpr*
OrTypePatternExpr	→	*UnaryTypePatternExpr*
	|	*OrTypePatternExpr* `||` *UnaryTypePatternExpr*
UnaryTypePatternExpr	→	*BasicTypePatternExpr*
	|	`!` *UnaryTypePatternExpr*
	|	`(` *TypePatternExpr* `)`
BasicTypePatternExpr	→	`void`
	|	*BaseTypePatternExpr DimsOpt PlusOpt*
DimsOpt	→	*DimsOpt* `[]` | λ
PlusOpt	→	`+` | λ
BaseTypePatternExpr	→	*PrimitiveTypePatternExpr*
	|	*NamePatternExpr*
PrimitiveTypePatternExpr	→	`boolean` | `...` | `double`
NamePatternExpr	→	*SimpleNamePatternExpr*
	|	*NamePatternExpr PackageSep SimpleNamePatternExpr*
SimpleNamePatternExpr	→	`*`
	|	`identifier`
	|	`identifierpattern`
PackageSep	→	`.` | `..`

Figure 2. Grammar for specifying context patterns in queries

The patterns of the query language were inspired in AspectJ join points. The main difference is that our language explicitly separates type patterns from execution point patterns. AspectJ users can gain even more rapid understanding of the pattern rules.

A *ContextPattern* is an expression that defines a type pattern. These expressions can use the wilcard '`*`'. Some examples of simple context patterns are:

- `com.my.Clazz`: matches with the type `Clazz` defined in the package `com.my`;
- `com.my.*Interface`: matches with any type that ends with `Interface` defined in the package `com.my`;
- `com..*`: matches with any type defined below package `com`.

The wildcard '`+`' can be used to select subtypes of the specified pattern. Some examples are:

- `java.io.InputStream+`: matches with any type that extends `java.io.InputStream`, including itself;
- `java.sql.Statement+`: matches with any type that implements `java.sql.Statement`.

The query language also allows composing context patterns using the logical operators `&&`, `||` or `!`, and grouping them with parenthesis. The wildcard `..` can be used to match inner types. Some examples are:

- `java.io.*Stream && !java.io.Output*`: matches with any type in package `java.io` that ends with `Stream` and do not start with `Output`;
- `*..*Filter || *..*Filtro`: matches with any type that ends with `Filter` or `Filtro` inside any package.

The detailed rules for context patterns are shown in Figure 2.

An *ExecutionPattern* matches an execution point. There are 4 types of execution point patterns: 1) method: the visibility modifiers are optional, and the return type,

ExecutionPattern → *OrExecutionPattern*
| *ExecutionPattern* `&&` *OrExecutionPattern*
OrExecutionPattern → *UnaryExecutionPattern*
| *OrExecutionPattern* `||` *UnaryExecutionPattern*
UnaryExecutionPattern → *BasicExecutionPattern*
| `!` *UnaryExecutionPattern*
BasicExecutionPattern → *BaseExecutionPattern*
| `(` *ExecutionPattern* `)`
BaseExecutionPattern → `?`
| `staticinitialization`
| *MethodPattern*
| *ConstructorPattern*
MethodPattern → *ModifiersOpt BasicTypePatternExpr*
MethodNamePattern `(` *FormalParametersOpt* `)`
ConstructorPattern → *ModifiersOpt* `new` `(` *FormalParametersOpt* `)`
MethodNamePattern → `*`
| `identifier`
| `identifierpattern`
ModifiersOpt → *ModifiersOpt Modifier* | λ
Modifier → `public` | `...` | `volatile`
FormalParametersOpt → *FormalParameters* | λ
FormalParameters → *FormalParameter*
| *FormalParameters* `,` *FormalParameter*
FormalParameter → `..`
| *BasicTypePatternExpr*

Figure 3. Grammar for specifying execution patterns in queries

method name and formal parameters are mandatory; 2) constructor: the visibility modifiers are optional, the reserved word `new` and the formal parameters are mandatory. The wildcard `..` is substitute for the latter; 3) class static initializer: the reserved word `staticinitialization` is mandatory; 4) `?`: wildcard that matches any of the above execution point. The detailed rules for execution point patterns are shown in Figure 3.

2.3. Architectural constraint language

The architectural constraint language is a subset of the query language. Logical composition operators for patterns are not present. This language enables the specification of constraints that will be used for checking architectural properties. There are two types of constraints, similar to the query language:

- Constraint on consecutive execution points: the execution point matched with $ExecutionPattern_r$ in the $ContextPattern_r$ *must be* followed by (considering only nodes in the execution tree) an execution point matched with $ExecutionPattern_c$ in the context matched with $ContextPattern_c$. The constraint is written as:

  `(`$ContextPattern_r$`,` $ExecutionPattern_r$`)` **`-->`** `(`$ContextPattern_c$`,` $ExecutionPattern_c$`)`

- Constraint on adjacent execution points: there must exist an execution point that matches with $ExecutionPattern_r$ in context $ContextPattern_r$, from which there is a linear path, in the execution tree, to an execution point that matches with $ExecutionPattern_c$ in context $ContextPattern_c$.

  `(`$ContextPattern_r$`,` $ExecutionPattern_r$`)` **`-+>`** `(`$ContextPattern_c$`,` $ExecutionPattern_c$`)`

The detailed rules the architectural constraint languages are not shown because of the lack of space, but they are similar to those of the query language, except for the

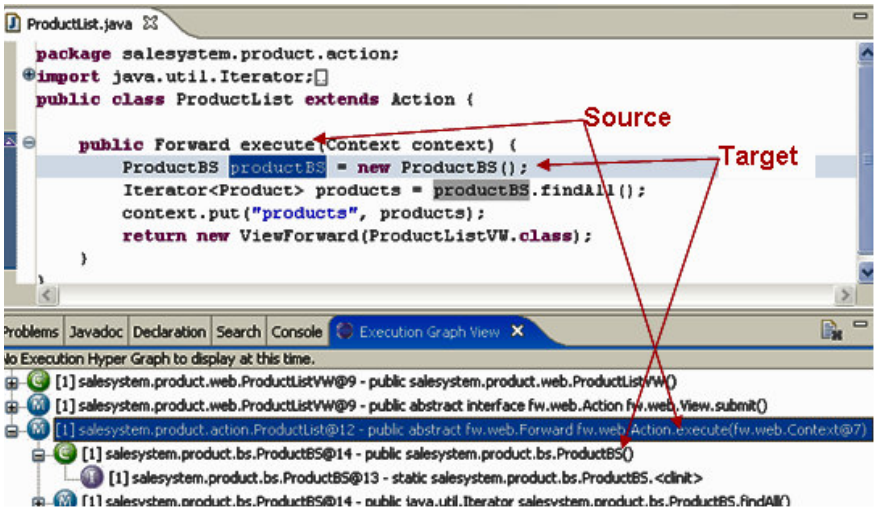

Figure 4. Key points of architectural constraint

absence of logical operators.

2.4. Annotation generator

The annotation generator is the component responsible for introducing annotations related to the key points of the source code that have architectural constraints. Key points are execution points that match with any architectural constraint specified with the architectural constraint language. An architectural constraint has two key points: source and target. Let be the following architectural constraint:

```
(fw.web.Action+, execute(..))  -+> (fw.bs.Business+, new())
```

The source point is defined as a call to the method `execute()` with any parameters, called from objects whose class extends the class `Action`, and the target point is a call to the constructor without parameters of any class that implements the interface `Business`. Figure 4 shows an instance of code that satisfies the constraint. The execution tree is shown in the lower part of the figure.

The annotation generator reads a file with the architectural constraints and writes a file with extension `.aj`. The generation process has to:

1. generate an aspect containing the specification of annotation types and respective declarations for system key points, containing
   (a) code for annotations types of the source key points;
   (b) code for annotations types of the target key points;
   (c) declaration of the annotations generated in source key points;
   (d) declaration of the annotations generated in target key points.

For the i-th architectural constraint in the file, the names of the annotations are generated as in the following table:

Execution Point type	Source annotation name	Target annotation name
Method	SourceMethodCall_i	TargetMethodCall_i
Constructor	SourceConstructorCall_i	TargetConstructorCall_i
Class static initializer	SourceStaticInitialization_i	TargetStaticInitialization_i
Wildcard "?"	SourceMethodCall_i SourceConstructorCall_i SourceStaticInitialization_i	TargetMethodCall_i TargetConstructorCall_i TargetStaticInitialization_i

It is necessary to generate annotation types for each key point because a key point may belong to several architectural constraints and Java allows only one annotation of same type for a key point type.

Consider the constraint `(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))`.

For step 1a, we have that source execution point is defined as `?`, so the three following annotation type declarations must be generated:

- Method:
```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface SourceMethodCall_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```
- Constructor:
```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface SourceConstructorCall_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```
- Class static initializer:
```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SourceStaticInitialization_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```

For the step 1b, the target execution point is a constructor, and thus will generate only one annotation type declaration

- Constructor:
```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface TargetConstructorCall_1  {
  String constraint() default
  "(fw.web.Action+, ?) -+> (fw.bs.Business+, new(..))";
}
```

For the step 1c, the generator will produce AspectJ `declare` statements for the annotation types generated in step 1a:

```
declare @method :
  * fw.web.Action+.*(..) :
  @SourceMethodCall_1();

declare @constructor :
  fw.web.Action+.new(..) :
  @SourceConstructorCall_1();

declare @type :
  fw.web.Action+ :
  @SourceStaticInicialization_1();
```

For the step 1d, the generator will produce the `declare` statement for the target annotation:

```
declare @constructor :
  fw.bs.Business+.new(..) :
  @TargetConstructorCall_1();
```

2.5. Conformance checking

The conformance checking takes place executing the program with two aspects:

- the aspect containing the annotation types and their declarations for the key points of the system, and

- the aspect of conformance checking.

Just like the execution aspect, the conformance checking aspect also builds the execution tree and after the execution, the aspect verifies if all constraints are satisfied in the execution tree. Since the constraint is written like a query, intuitively, a constraint is satisfied if the result of the corresponding query is non-empty. In other words, a constraint `source --> target` is satisfied if for some matched `source` there exists a matching `target` as a immediate son in the execution tree, and a constraint `source -+> target` is satisfied if for some matched `source` there exists a matching `target` that can be reached below in the execution tree.

3. Application

This section presents some guidelines to recover architectural restrictions and we apply them on two case studies: RegSys e CUP.

3.1. Guidelines to recover architectural constraints

This section presents guidelines that helps the developers to recover important architectural constraints from the source code.

The first step is obtaining artifacts that will help to understand the main components and conectors of the system architecture: the source code that will be used to execute the system weaved with the Execution Aspect that builds the execution tree; the system user guide that helps the developer undertanding the main system entities and the functionalities related to those entities; and the system technical specifications, that also help identifying the main components of the system.

The next step is to execute the system with the Execution Aspect that intercepts the execution points and creates the execution tree. It is recommended to execute a scenario that covers the main use cases.

Once the execution tree is created, the next step is to analyze this tree aided by the query language presented before. Some actions are recommended:

1. The expansion of the execution tree analyzing the number of threads.
2. The execution of queries that filter interesting information about the main components of the system:
    - When instances of these components are created:

      ```
      (java.lang.Thread, public void run()) -+> (*.*MainComponent1 ||
      *.*MainComponent2, new(..))
      ```
    - Which classes use these components:

      ```
      (*, ?)  --> (*.*MainComponentX, ?)
      ```
    - Which classes these components use:

      ```
      (*.*MainComponentX, ?)  --> (*, ?)
      ```
    - Data input and output (files, databases, etc):

      ```
      (*, ?)  --> (java.io.File, new(..))
      ```
    - The main conectors of the architecture:

      ```
      (*,?)  --> (java.io.PipedWriter, void connect(java.io.PipedReader))
      ```
3. The definition of system architecture using some architecture description language, such as, Acme[Garlan et al. 2000] or UML.
4. The definition of the architectural constraints using the queries that helped finding the conectors of the architecture.

3.2. Case study 1: Regsys

The system `RegSys`[1] was presented in [Yan et al. 2004]. `RegSys` has architectural style *Pipes* and *Filters*. It is a small application with 4 classes and 355 LOC. For the sake of understanding, follows an overview of the classes:

- `SplitFilter`: reads data of one student in the input file and verifies if his course is Computer Science. If so, the data is sent to a pipe, else the data is sent to another pipe.
- `PassFilter`: verifies if the student has prerequisites to enroll on courses. If so, the student data is sent to the next pipe, else the data is discarded.
- `MergeFilter`: reads the two data flows of the two `PassFilter`s and writes a result file.
- `RegSys`: creates and initialize execution of filters. The input and output file path is provided for the main method.

After obtaining the source and analyzing the documentation in [Yan et al. 2004], the next step is to execute the system with the execution aspect to obtain the execution tree. The execution scenario is the simplest one: just run without user interaction.

During the exploration of the execution tree, five threads are detected, each one with its respective execution tree.

- `main: [1] java.lang.Thread@1 - public void run()`
- `SplitFilter: [1] v1.SplitFilter@20 - public void run()`
- `PassFilter: [1] v1.PassFilter@27 - public void run()`
- `PassFilter: [1] v1.PassFilter@31 - public void run()`
- `MergeFilter: [1] v1.MergeFilter@33 - public void run()`

We can see that *Filter* components ends with suffix `Filter`. Initially we can suppose these components were created in the main thread. This can be verified with the following query:

```
(java.lang.Thread, public void run()) -+> (*.*Filter, new(..)).
```

The query actually returns the creation of all components that ends with `Filter`. An important observation is that two objects `PassFilter` were created. The next step can be the identification of the input and output files. The following query identifies what `java.io.File` objects are created during program execution:

```
(*, ?)  --> (java.io.File, new(..)).
```

The query returns two files created during the execution of the method `void main(String[])`: `java.io.File@5` and `java.io.File@7`.

At this point, we do not know if they are input or output files. Another query can be issued to identified the file roles: `(*, ?)  --> (*, new(java.io.File))`.

Figure 5 shows the results of the query. The first file, `java.io.File@5`, is used for data entry because it is used to create an `java.io.FileReader` object; the creation of the `FileReader` object occurs during the execution of the method `void run()` of the component `v1.SplitFilter@20`. This is a strong indication that this component reads data from the input file; the second file `java.io.File@7`, has a data output role because is used to create an `java.io.FileWriter` object; the creation of the `FileWriter` object occurs during the execution

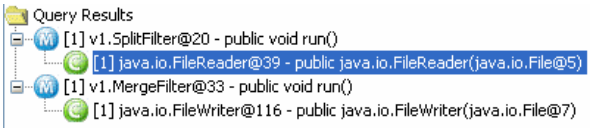

Figure 5. Result for `(*, ?) --> (*, new(java.io.File))`

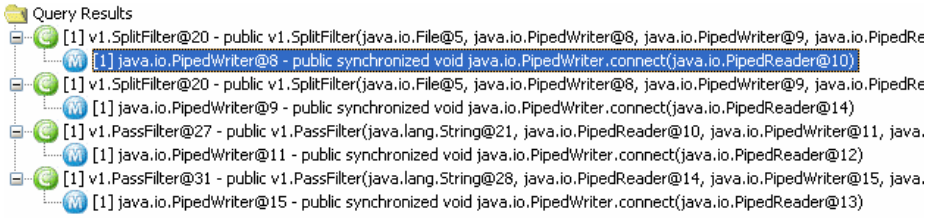

Figure 6. Result for `(*,?) --> (java.io.PipedWriter, void connect(java.io.PipedReader))`

of the method `void run()` of the component `v1.MergeFilter@33`, strongly indicating that this component writes the output file data.

The hypothesis that the component `v1.SplitFilter@20` reads the input file could be confirmed with the query:

```
(v1.SplitFilter, public void run()) -+> (java.io.FileReader, * read*()).
```

Unfortunely, this query has no results. We can rewrite it more generally:

```
(v1.SplitFilter, public void run()) -+> (java.io.Reader+, * read*()).
```

Now the query returns a result because the reader object is from class `java.io.BufferedReader@40` and not from `java.io.FileReader`. A similar query can be used to confirm that the component `v1.MergeFilter@33` writes the output file:

```
(v1.MergeFilter, public void run()) -+> (java.io.Writer+, void write*(..)).
```

The next steps are related with the recovery of conectors. In the execution tree can be found some objects `java.io.PipedWriter` and `java.io.PipedReader`, probably having roles of a pipe. The following query identifies the creation of these objects:

`(*, ?) --> (java.io.PipedWriter || java.io.PipedReader, new())`. The query returns the several objects of `PipedWriter` and `PipedReader` classes, created during the execution of `void main(String[])`.

The next query identifies which `java.io.PipedReader` objects were conected to `java.io.PipedWriter` objects:

```
(*,?) --> (java.io.PipedWriter, void connect(java.io.PipedReader)).
```

The results of this query are shown in Figure 6. During the creation of the component `SplitFilter@20`, the object `java.io.PipedReader@10` is conected to the object `java.io.PipedWriter@8` and the object `java.io.PipedReader@14` is conected to the object `java.io.PipedWriter@9`. These facts are evidences that the component has two output ports; during the creation of the component `PassFilter@27`, the object `java.io.PipedReader@12` is conected to the object `java.io.PipedWriter@11`. So this component has one output port. during the creation of the component `PassFilter@31`, the object `java.io.PipedReader@13` is

[1] Available in: http://able.fluid.cs.cmu.edu:8080/Able/DiscoTect/PipeFilterExample.zip

conected to the object `java.io.PipedWriter@15`. This component has also one output port.

The next query shows that the component `SplitFilter@20` writes in the two output ports:

```
(v1.SplitFilter, ?)  -+> (java.io.PipedWriter, * write(..)).
```

The result of the query confirms this showing that the method `write` is called in context of objects `java.io.PipedWriter@8` and `java.io.PipedWriter@9`. Now it is necessary identify from where components `v1.PassFilter` read their data. The following query can be used:

```
(v1.PassFilter, ?)  -+> (java.io.PipedReader, * read(..)).
```

The result show that is `java.io.PipedReader@14`.

The next query shows that `PassFilter` components writes in their output ports:

```
(v1.PassFilter, ?)  -+> (java.io.PipedWriter, * write(..)).
```

Since this query is not null, we have the confirmation. Now, we need to know from where the component `v1.MergeFilter@33` reads its data:

```
(v1.MergeFilter, ?)  -+> (java.io.PipedReader, * read(..)).
```

The result shows that is `java.io.PipedReader@12`. We conclude that: there is a pipe between the components `PassFilter@27` and `MergeFilter@33`, because the object `java.io.PipedReader@12` was conected to the object `java.io.PipedWriter@11` and the component `MergeFilter@33` reads its data from port `java.io.PipedReader@12`; and there is a pipe between the components `PassFilter@31` and `MergeFilter@33`, because the object `java.io.PipedReader@13` was conected to the object `java.io.PipedWriter@15` and the component `MergeFilter@33` reads its data from port `java.io.PipedReader@13`.

The complete description of the recovered architecture was written in Acme and shown in Figure 7 and is equivalent to the description presented in [Yan et al. 2004]. Some possible architectural constraints that can be imposed on this architecture are derived from the conectors of the architecture. These constraints, if satisfied in future software changes, will preserve the *Pipe-Filter* architectural style.

```
(*.SplitFilter, void run()) -+> (java.io.FileReader+, * read*())
(*.MergeFilter, void run()) -+> (java.io.Writer+, void write*(..))
(*.SplitFilter, ?)  -+> (java.io.PipedWriter, * write(..))
(*.PassFilter, ?)  -+> (java.io.PipedReader, * read(..))
(*.PassFilter, ?)  -+> (java.io.PipedWriter, * write(..))
(*.MergeFilter, ?)  -+> (java.io.PipedReader, * read(..)).
```

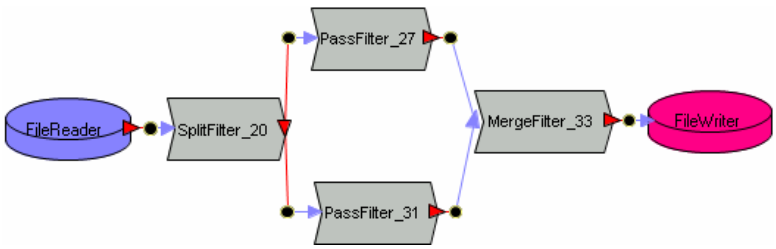

Figure 7. Architectural description of `RegSys`

3.3. Case study 2: CUP

This section summarizes a case study of recovering architectural constraints of CUP, an LALR parser generator. The implementation has 40 classes and 5623 LOC.

The main class of `CUP` was executed with the Execution Aspect for a simple grammar and the execution tree was generated. Only one thread was created. The main events in the executin tree are:

1. `(java_cup.Main@2, void parser_args(java.lang.String[]@3)`: this method receives the same arguments of the `main`.
2. `(java.io.BufferedInputStream@32, BufferedInputStream( java.io.InputStream@26)`: this event is a strong indicative that the object `BufferedInputStream@32` is the stream of the input file. The following query confirms this hypothesis:
   `(java_cup.Main, void main(java.lang.String[])) -+> (java.io.InputStream+, new(..)).`

The identification of the responsible component for reading the input file can use the following query: `(*, ?) --> (java.io.InputStream+, * read*(..))`

The result shows that the component `java_cup.lexer@3711` is responsible for reading the file, and the empty result of the following query shows that this is the only component responsible for reading:

```
(!java_cup.lexer, ?) --> (java.io.InputStream+, * read*(..)).
```

The identification of a client for the lexer can use the following query:

```
(*, ?) --> (java_cup.lexer, ?).
```

Many results are shown, including:

- `java_cup.parser@3705`;
- `java_cup.lexer@3711`.

Thus, the `java_cup.lexer@3711` invoke its own methods. The following query reduces the result for analysis:

```
(!java_cup.lexer, ?) --> (java_cup.lexer, ?).
```

The new result reveals an important relationship between `java_cup.parser@3705` and `java_cup.lexer@3711`: the call to `java_cup.runtime.Symbol next_token()`. Thus, we can conclude that java_cup.parser@3705 is the parser component.

The following query identifies file writing in Cup:

```
(*, ?) --> (java.io.OutputStream+, new(..)).
```

The result shows the creation of output streams for 2 files: `java.io.File@21227` and `java.io.File@21235`. Supposing files would be written with the API of `java.io.OutputStream` we could try the following query:

```
(*, ?) --> (java.io.OutputStream+, void write*(..)).
```

However, the result is empty. Another try would be `java.io.Writer`:

```
(*, ?) --> (java.io.Writer+, void write*(..)).
```

Again, the result is empty. So, we can try to discover which constructor call received instances of `java.io.OutputStream` as parameters:

```
(*, ?) --> (*, new(java.io.OutputStream+)).
```

The results shows that was a `java.io.PrintWriter` constructor. Now, the identification of the component responsible for writing output files can use the query:

```
(*, ?)  --> (java.io.PrintWriter, * print*(..)  || * write*(..)).
```

The results shows that `java_cup.emit@27` writes output files. The following query shows that it is the only component to write output files, because its result is empty.

```
(!java_cup.emit, ?)  --> (java.io.PrintWriter, * print*(..)  || * write*(..)).
```

The architecture of CUP is shown in Figure 8. Indeed, architectural constraints for CUP can be derived from the connectors, similarly to the RegSys case.

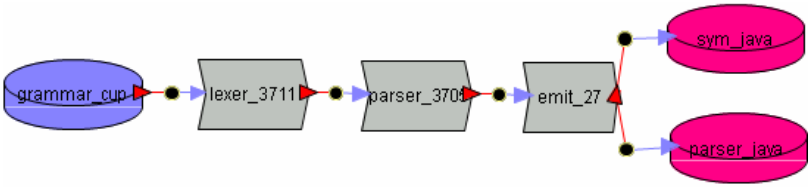

Figure 8. Architecture of CUP

4. Discussion and Related work

Several results have already been achieved in architecture recovery and dynamic analysis of software [Feijis et al. 1998, Safyallah and Sartipi 2006, Xiao and Tzerpos 2005, Vasconcelos and Werner 2005, Gorton and Zhu 2005].

Our work is more related to Discotect [Yan et al. 2004] and ArchJava [Aldrich et al. 2002]. The recovery of architectural properties was inspired in Discotect: the creation of a state machine to observe the creation of objects and their interaction using method calls. However, instead of creating a state machine, we propose a query language with operators similar to those of AspectJ. We highlight the operator `+` that verifies if the context of an execution point is subtype of the specified pattern, and the wilcard `?` that matches with any execution points. Such operations are not available in Discotect.

Another contribution compared with Discotect is that after executing the general scenario of the system, the user can specify queries about the system execution. This facilitates refinements of execution tree to identify the relevant events for the system architecture. In Discotect is necessary to build a new state machine for each refinement step and then reproduce the same scenario execution. This can be an arduous task.

Our proposed guidelines for identifying important architectural events suppose that the user will spend some time investigating the documentation about the main components of the system. This information will help to produce better queries. Since Discotect does not allow query definition, it does not make this assumption.

Likewise Discotect, our solution is strongly tied to the names of components and operations. Changing names in the source code can invalidate a query that identifies an architecture component.

An important tool for conformance checking is ArchJava. Although ArchJava is a new programming language and thus provide more expressiveness, our approch is more pragmatic because it is not strongly coupled and intrusive as ArchJava. In our approach the architectural properties are not scattered inside the source code. Our architecture is

specified in a separate higher-level document, containing only the main components and conectors of the system. Our approach uses a custom language to specify architectural constraints that is a subset of the query language. This facilitates specifying constraints on systems that had their architecture recovered with execution queries. This separation also enables two types of development roles: the architect role that writes architectural constraints, generate annotations in the key execution points and verifies the conformance of system, and developer role that has to implement the system according to the architecture properties specified by the architect.

5. Conclusions

The software architecture has an important role both for the project and for the development and maintenance of software. Moreover, several software systems are implemented without having a predefined architecture. In other situations, the architecture of the system was designed in advance, but may be outdated in relation to actual implementation. This work presents the following solution to these problems:

1. a language to assist in the recovery of the software architecture of programs written in Java. The language enables queries in the execution tree of the program;
2. a methodology for recovering the software architecture using the tool above;
3. a language to specify architectural constraints between components of the software architecture description;
4. a mechanism to verify conformance between the architecture and implementation of the system.

Future work includes developing automatic techniques to generate interesting queries and automatically generating the system architecture, performing case studies in larger systems, and formal specification of the query and constraint languages.

References

Aldrich, J., Chambers, C., and Notkin, D. (2002). Archjava: connecting software architecture to implementation. In *Proc. of ICSE '02*, pages 187–197.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 1st ed.

Feijis, L., Krikhaar, R., and Ommering, R. V. (1998). A relational approach to support software architecture analysis. *Software – Practice and Experience*, 28(4):371–400.

Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, pages 47–68. Cambridge.

Gorton, I. and Zhu, L. (2005). Tool support for just-in-time architecture reconstruction and evaluation: An experience report. In *ICSE - Intl. Conf. on Software Engineering*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *ECOOP '01*.

Safyallah, H. and Sartipi, K. (2006). Dynamic analysis of software systems using execution pattern mining. In *Proceedings of the 14th IEEE Intl. Conf. on Program Comprehension (ICPC'06)*.

Vasconcelos, A. and Werner, C. (2005). Um conjunto de heurísticas de agrupamento de classes para apoiar a recuperação da arquitetura de software. In *Proc. of WMSWM'2005*, pp. 34–49.

Xiao, C. and Tzerpos, V. (2005). Software clustering based on dynamic dependencies. In *Proc. of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*.

Yan, H., Garlan, D., Schmerl, B., Aldrich, J., and Kazman, R. (2004). Discotect: A system for discovering architectures from running systems. In *Proc. of ICSE '04*, pages 470–479.