# Experiments with Pro-active Declarative Meta-Programming

Verónica Uquillas Gómez
Software Languages Lab
Pleinlaan 2
1050 Brussel
Vrije Universiteit Brussel
vuquilla@vub.ac.be

Andy Kellens
Software Languages Lab
Pleinlaan 2
1050 Brussel
Vrije Universiteit Brussel
akellens@vub.ac.be

Kris Gybels
Software Languages Lab
Pleinlaan 2
1050 Brussel
Vrije Universiteit Brussel
kris.gybels@vub.ac.be

Theo D'Hondt
Software Languages Lab
Pleinlaan 2
1050 Brussel
Vrije Universiteit Brussel
tjdhondt@vub.ac.be

## Abstract

Program querying has become a valuable asset in the programmer's toolbox. Using dedicated querying languages, developers can reason about their source code in order to find errors, refactoring opportunities and so on. Within Smalltalk, the SOUL language has been proposed as one such language that offers a declarative and expressive means to query the source code of object-oriented programs.

Ever since its inception, SOUL has been used as the underlying technique for a number of academic software engineering tools. Despite its success, one of the problems of SOUL is that, due to its backward chained implementation, it is less suited as a basis for such pro-active software tools. Using SOUL, a developer has to launch the queries over the system manually, rather than automatically receiving feedback whenever the underlying source code is changed. In this paper we present PARACHUT, an alternative logic query language that is based on forward chaining and temporal logic and that allows developers to express queries over the change history of the system. Furthermore, PARACHUT's data-driven nature makes it possible to provide instant feedback to developers when the source code is changed, thus providing better support for pro-active software tools.



## 1. Introduction

Declarative meta-programming is the use of a declarative programming language in order to reason about the implementation of software. Over the last ten years, this technique has been well studied at the Vrije Universiteit Brussel and has led to the development of SOUL [19], the Smalltalk Open Unification Language. SOUL is a PROLOG-like language that exists in symbiosis with the underlying Smalltalk language and that provides libraries of predicates to reason over Smalltalk programs.

From its inception, SOUL has been used as a research vehicle for a wide range of applications, such as architectural verification [13], co-evolution of design and implementation [14] and aspect-oriented programming [6]. Further development of the language has resulted in extensions to the language such as template queries [1] and the implementation of libraries of predicates to support reasoning over other programming languages such as Java, C(++) and Cobol.

One of the key characteristics of SOUL is that it is a backward-chained language, meaning that program queries that are written down in SOUL are resolved in a demand-driven way: a user of the language has to launch a query in order to retrieve all the results. This does not always align well with the application domain of SOUL, namely as an underlying platform for building software engineering tools. First, this backward chaining results in that the same query is always calculated again from scratch. For time consuming queries, this can result in a serious overhead. Second, the demand-driven nature of SOUL does not fit well with the pro-active nature of some software engineering tools. Rather than having to launch queries over the system,

these tools tend to work pro-actively, providing developers with feedback whenever it becomes available.

In this paper we relate our experiences implementing and using PARACHUT (**P**rogramming **a**nd **R**easoning **a**bout **Ch**anges **u**sing **T**ime). PARACHUT is an alternative instantiation of the concept of declarative meta programming that aims at alleviating the above problems. PARACHUT differs from SOUL in two important ways. Firstly, rather than working demand-driven PARACHUT reacts on changes in the development environment in order to trigger the reasoning process over software. Secondly, it offers the developer an immediate feedback of the reasoning process. To this end, PARACHUT offers a forward chaining logic inference engine implemented in Smalltalk. An additional advantage of this forward chaining is that logic queries can be processed incrementally rather than having to be computed from scratch over and over again.

Furthermore, PARACHUT is not only triggered by changes in the development environment, these changes are also reified into the knowledge base over which we can reason about using PARACHUT. As a result, a user of the language can not only access the current state of the system, but can also query the change history that is built up during the development process. In order to ease this reasoning over the history of changes, PARACHUT incorporates temporal logic operators into the query language.

## 2. Logic Meta-Programming

Logic Meta-Programming is the use of a logic programming language in order to reason about software systems. Key to logic programming languages is that they are declarative: rather than having to specify *how* one wants to calculate a certain query over the source code of the system, one specifies *what* one wants. The inference engine of the logic programming language will then resolve the query and return the results to the user. As a result, logic programming languages are very suitable for writing *meta-programs* that reason over software [3, 14].

In the past, logic meta-programming has successfully been applied in software engineering to support problems such as the co-evolution of design and implementation [3, 14], or as a basis for aspect-oriented programming languages [8, 6].

As a particular instantiation of logic meta-programming, the SOUL (Smalltalk Open Unification Language) has been developed [19]. SOUL is an implementation of PROLOG in Smalltalk, that offers a tight symbiosis with the underlying Smalltalk system and that relies heavily on Smalltalk's reflective capabilities in order to reason about programs.

On top of the SOUL language, libraries of predicates have been built that allow for reasoning over Smalltalk, Java, C(++) and Cobol programs. We illustrate SOUL's syntax by means of one example, namely the implementation of the logic predicate `classInHierarchyOf` (see also Table 1) that verifies whether one class is present in the inheritance hierarchy of another class.

This rule is defined as:

```
1  classInHierarchyOf(?dirSubClass, ?superClass) if
2     subclassOf(?dirSubClass, ?superClass)
3
4  classInHierarchyOf(?indSubClass, ?superClass) if
5     subclassOf(?dirSubClass, ?superClass),
6     classInHierarchyOf(?indSubClass,?dirSubClass)
```

The above predicate consists of two rules. Without going into details, the first rule states that a class is present in the hierarchy of another class if it is a direct subclass of that class. The second rule works for the transitive case: there might be indirect subclasses underneath the root of the hierarchy. SOUL's syntax only differs slightly from PROLOG's, namely that SOUL uses the keyword `if` instead of `:-` and logic variables are identified by a question mark (e.g. `?superClass`) instead of being capitalized.

As mentioned above, SOUL is tightly integrated with the underlying Smalltalk language, resulting in a symbiosis between both languages. This symbiosis makes it possible to use Smalltalk code within SOUL queries, either as a logic condition or as a value that will be used in the computation. In combination with the reflective capabilities of Smalltalk, this easily allows us to query the classes, methods, ... that are present in the Smalltalk image. For example, the logic predicate `class` that retrieves all classes in the Smalltalk image (or that verifies whether its argument is a class) is defined as follows:

```
1  class(?class) if
2    member(?class, [Smalltalk allClasses])
```

Smalltalk blocks included into SOUL queries are indicated by means of square brackets (`[ ]`). In the implementation of the `class` predicate, the set of all classes is computed by means of the Smalltalk expression `Smalltalk allClasses`. This expression returns a collection of all the classes that are present in the Smalltalk system. The `class` predicate is defined then as all members of this collection of classes.

Finally, SOUL is a backward chained language. This means that the triggering of the resolution process is done explicitly by means of the user of the language. In other words, as soon as the user asks SOUL the answer to a particular query, the inference engine will retrieve all possible solutions for that query. For example, consider the following query:

```
1  if classInHierarchy(?class, [Collection])
```

Queries in SOUL are indicated by starting with the keyword `if`. The query above will query the Smalltalk image and provide bindings for the logic variable `?class` for all possible classes that are present in the hierarchy of the `Collection` class.

| Rule | Description |
|---|---|
| class(?class) | ?class is a class |
| metaClass(?class, ?metaClass) | ?class has meta-class ?metaClass |
| instanceVariableInClass(?class, ?instVar) | ?class has instance variable ?instVar |
| methodInClass(?class, ?method) | ?method belongs to ?class |
| methodCallsMethod(?class, ?method, ?methodR) | ?method in ?class calls ?methodR |
| methodReferencesClass(?class, ?method, ?classR) | ?method in ?class references ?methodR |
| methodUsesInstVariable(?class, ?method, ?instVar) | ?method in ?class accesses ?instVar |
| namespace(?nameSpace) | ?nameSpace is a namespace |
| classInNamespace(?class, ?nameSpace) | ?class belongs to ?nameSpace |
| namespaceInPackage(?namespace, ?package) | ?nameSpace belongs to ?package |
| package(?package) | ?package is a package |
| classInPackage(?class, ?package) | ?class belongs to ?package |
| bundle(?bundle) | ?bundle is a bundle |
| packageInBundle(?package, ?bundle) | ?package belongs to ?bundle |
| subclassOf(?subClass, ?superClass) | ?subClass is subclass of ?superClass |
| classInHierarchyOf(?dirSubClass, ?superClass) if subclassOf(?dirSubClass, ?superClass)<br>classInHierarchyOf(?indSubClass, ?superClass) if subclassOf(?dirSubClass, ?superClass), classInHierarchyOf(?indSubClass,?dirSubClass) | ?dirSubClass and ?indSubClass classes are in the hierarchy of ?superClass |

**Table 1.** Logic Predicates expressing the entities and relationships of *Program Changes*

# 3. PARACHUT

PARACHUT (**P**rogramming **a**nd **R**easoning **a**bout **Ch**anges **u**sing **T**ime) is inspired by the idea of providing support to developers during the software development process by reasoning about past and current program changes.

In this paper, we propose a code assistant tool and program query language that provides support that goes beyond the analysis of the current implementation of a system, that performs an online analysis of software implementation at the development time in order to react to each program change, and thus assists the developers in accomplishing their tasks.

Our approach is an application of declarative meta programming combined with temporal logic, and uses a forward chainer as a reasoning engine to pro-actively provide feedback to the developers. The integration of those approaches provides a powerful medium for reasoning about (past) program changes.

## 3.1 Forward Chaining

We selected forward chaining as the underlying technique to support reasoning over the source code of a system, and the changes in such a system. Forward chaining is a data-driven reasoning process, which generates all the possible solutions from initial data. Typically, forward chainers are implemented by means of the Rete algorithm [5]. While a discussion of this algorithm lies outside the scope of this paper, intuitively this algorithm works by representing a set of logic rules as a network of nodes. At each level in the network, a cache is used to store the intermediate results for particular facts and rules. Whenever a new fact is added to the reasoning engine, this fact is percolated throughout the network and possible new conclusions of rules are reported. While the initial starting cost of setting up the Rete network can be quite expensive, and Rete networks are known to consume lots of memory, the large advantage of this technique is that changes in the fact base (in our case the changes performed on the system) immediately and efficiently trigger the calculation of new conclusions based on the rules. Furthermore, due to the caching in the Rete network, intermediate results of computations are cached such that the inference engine does not need to repeatedly calculate them.

The data domain of our problem are the source-code entities in the system that are represented by the changes that are made to them. Each time a change is made, data representing such a change is provided to the inference engine. Our rules reason over the source-code entities and changes, and are used to infer new data whenever is possible. This new data can in turn infer more data, and so on.

The above fundamental differences between the forward chaining that underlies PARACHUT and the backward chaining of SOUL result in that PARACHUT is an ideal candidate reasoning engine to use as a basis for software engineering tools that need to react immediately to changes in the source code and pro-actively support their users with feedback.

## 3.2 Temporal Logic Programming

The term *Temporal Logic Programing* [15] refers to a subset of logic programming that has been used to describe approaches that represent temporal information within a logical framework. *Temporal Logic* abstracts the explicit handling of time and queries are evaluated to an implicit temporal context. PARACHUT makes use of this general definition of Temporal Logic Programming in order to reason about the current program changes against *past* program changes.

Temporal Logic Programming extends a logic programing language with temporal predicates that reason about the

time information. They refer to past time (e.g. *previous*), to the future (e.g. *next*), and others refer to past and future time simultaneously (e.g. *since*). In our approach, we support some of those temporal operators, mainly past operators such as: `mostrecent` and `allpast`, and `since`. These predicates are supported by integrating temporal reasoning with our Rete network implementation. More concretely, we extended the set of possible nodes in the Rete network with temporal nodes that explicitly handle the time information and make it accessible to the temporal predicates.

### 3.3 Stages of our Approach

***Specifying the Declarative Meta Programs***

PARACHUT reasons about the development process of an object-oriented software system written in Smalltalk, specifically about *program changes*.

As a first step we need to write the *declarative meta-programs* in charge of describing and reasoning about the *program* and the *program changes* made by a developer while he is working on the source code. The logic rules and facts describe the *program changes* (our domain) and the process of reasoning about those changes.

The syntax and primitive predicates of PARACHUT (a subset of which are listed in Table 1, along with an explanation of the predicates) are as close to SOUL's as possible. Existing libraries of SOUL rules can therefore be re-used in PARACHUT as well. The exception is that a few special primitive predicates, and some syntactic sugar that is provided for them in SOUL, are not supported in PARACHUT. These predicates provide a reflective facility for controlling the backward chaining process. Naturally they can not be supported by a forward chainer. Use of these predicates in SOUL, and in PROLOG in general, is discouraged and it is strongly recommended to only use them for optimizations than can be easily removed.

In Figure 1, we present the library of predicates that was written for the creation of our Rete network. As for tool support, we leverage part of the user interface of SOUL in order for a developer to write logic rules.

PARACHUT offers two different types of predicates. A first type of predicate reasons over the actual source-code entities that are present in the system at any given moment in time. These predicates allow a user to express queries that are similar to those expressed using the SOUL language. However, since we track the changes that are made to the system, we have more fine-grained information about the history of the source code. In order to reason about these changes, and the explicit time model they impose, PARACHUT offers also a set of temporal predicates that reason about the changes in the program and the temporal relationships between these changes. To implement this, we extended the Rete forward chaining algorithm with a *Temporal Rete* component that supports such temporal reasoning. Therefore, since we have incorporated the notion of time

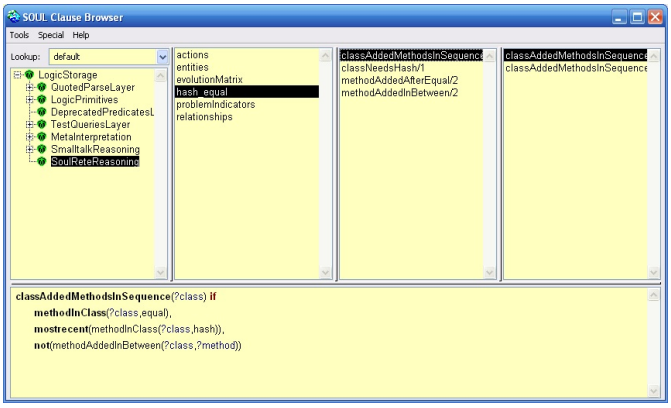


**Figure 1.** The library of logic rules that are used to create our Rete network

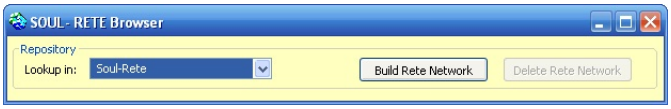


**Figure 2.** Soul-Rete Browser

by means of the temporal predicates, our *declarative meta-programs* are also *temporal declarative meta-programs*.

A *program change* is in turn represented by the entities and relationships being modified in the source code, such as classes, methods, instances variables, super classes, and so on. PARACHUT does the reification of the *program changes* by means of *logic facts*. They correspond to the set of facts entered into, or retracted from the Rete network and the set of facts being inferred by it.

***Activating the Facts Reasoner***

The component which is playing the role of facts reasoner is the *Temporal Rete*. As mentioned above, it is a temporal logic forward chaining inference engine that extends the original Rete algorithm to produce a reasoner capable of manipulating logic rules and a new set of temporal and non-temporal nodes in the Rete network.

The *Temporal Rete* is responsible for parsing the logic rules and for compiling them into a Rete network. Once the network is created the inference engine is ready to perform the reasoning process. Figure 2 shows the basic interface used by a developer to create or remove a Rete network.

***Tracing Changes***

One of the components integrated in PARACHUT is SmallBrother, a meta-program written in VisualWorks Smalltalk by Roel Wuyts. SmallBrother observes the source code of the underlying Smalltalk system and reports its clients of changes in this source code. Within PARACHUT, this meta-program is in charge of tracing the changes made by a developer for a further generation of the facts representing such changes.

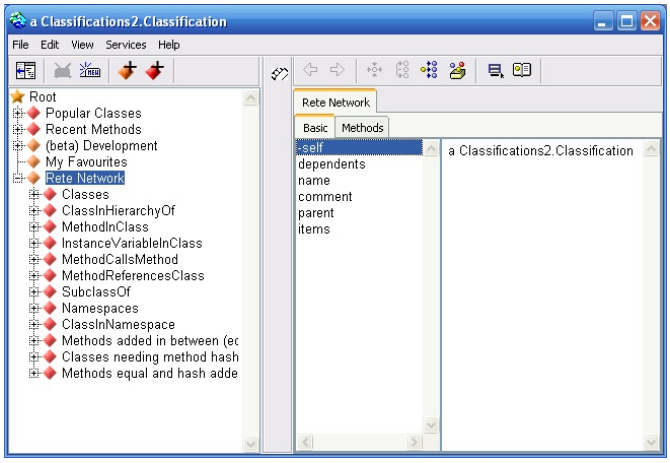


**Figure 3.** The Rete classification defined in StarBrowser

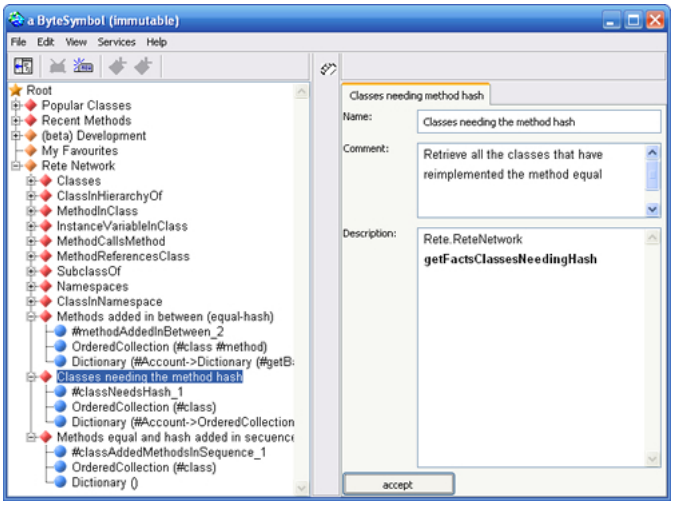


**Figure 4.** The classifications created in *StarBrowser* to explore the `Hash - Equals` example

We receive the data captured by the tracer, such as the type of change, and the objects involved in the change. Next, we invoke our programs which generate the logic facts mapping that data.

***Creating and Adding Facts to the Rete Network***

For each kind of change (addition, deletion or modification), we create or select facts expressing those changes. Our approach supports two representations of the program changes as logic facts: changes expressed by the presence of the entities (.e.g `class(?class)`) and changes expressed by the action performed by the developer (e.g. `addedClass(?class)`). Although these facts are similar they have a different meaning that impacts on the inference process.

One implication of the use of temporal logic programming is that each fact in the Rete network contains a time interval for which it is considered true. When a fact which represents an action is inserted in the network (e.g. `addedClass`) it is considered true only at that point in time. In other words, the fact of the example adding a class is only true at the point in time at which the action was performed. On the other hand, when a fact which represents the *presence* of an entity (e.g. `class`) is inserted in the network, it is considered true from that point in time until $\infty$. When the entity is removed from the system, its corresponding fact's end time will correspond to the time at which the entity was removed.

The process of adding facts ends when there are no more facts to be processed. We want to emphasize that not only the facts that arise from changes in the source code have an impact on the results of the network. Since during the reasoning process new facts are derived, these facts also get inserted into the Rete network and can in turn result in that other facts gets deduced, and so on.

***Presenting Outputs to Developers***

Our main goal was to develop an integrated (prototype) of a tool suite focusing on the underlying technology and not on the presentation of output to developers. The facts resulting from the reasoning process of the *program changes* are stored in the *Rete Network*, and they are presented to developers by means of the StarBrowser [20].

The StarBrowser is an easy customizable source-code browser for VisualWorks. The goal of the StarBrowser is to allow browsing the Smalltalk environment, classifying items (objects, classes, methods, packages, etc.) and manipulating these classifications and their contents using different kinds of editors. The classifications are synchronized with the information they provide. Other tools are also integrated to or in some cases extended the StarBrowser. Examples of such integrations are: SOUL, IntensiVE [14], CodeCrawler [10], CoBro-Nav [2].

Using this integration with the StarBrowser, a user of PARACHUT can access the information in the Rete network, and browse the nodes of the network (i.e. the logic rules). Users can select which rules (nodes) they are interested in. For each of these selected rules, a classification is added containing the elements that belong to the rule. Whenever new results for these rules percolate out of the network, they will be added to the corresponding classification in the StarBrowser (see Figures 3 and 4).

## 4. Example queries

In this section, we present some example queries using PARACHUT to demonstrate its utility as a pro-active code assistant reasoning about *program changes* constrained to past time. The combination of declarative meta programming with temporal logic provides a mechanism to write down more expressive logic rules capable of reasoning about changes using time information.

The examples are classified in two groups. The first group of examples is based on the Evolution Matrix [9] approach. We provide representations of some of the patterns for classes proposed by the Evolution Matrix. The second group of experiments represent indicators for detecting (possible) problems in the development process. They are applied to small cases which allow to illustrate how the logic program are enhanced by using temporal logic.

### 4.1 Evolution Matrix: shape-based patterns

The first two examples we show here are based on the Evolution Matrix [11, 9] proposed by Michele Lanza. The Evolution Matrix is a polymetric view that supports the understanding of the evolution of the classes of a software system and the evolution of the system itself. Although the Evolution Matrix is more suited to study the evolution of a system over an extended number of versions in order to provide a macroscopic view of the system's evolution, we here illustrate the use of PARACHUT on a smaller scale, by finding interesting patterns in the fine-grained changes that can be tracked using PARACHUT and that are inspired by the Evolution Matrix patterns.

Using the Evolution Matrix, classes are categorized based on the recurrent patterns observed in the matrix during their evolution. One category is **Shape-based patterns**. These patterns depend on the growing or shrinking of a class during the different stages of the implementation. For this category five types of classes were proposed, and we present the examples of two of them: *Pulsar* and *Supernova*.

The Evolution Matrix uses metrics in order to establish each type of classes proposed. Such as the number of methods of a class, the number of added methods between versions, etc. Figure 5 depicts a schematic Evolution Matrix. A column represents a version of the software. A row represents the different versions of the same class. Finally, the order of the classes in each column represents the order in which they were created.

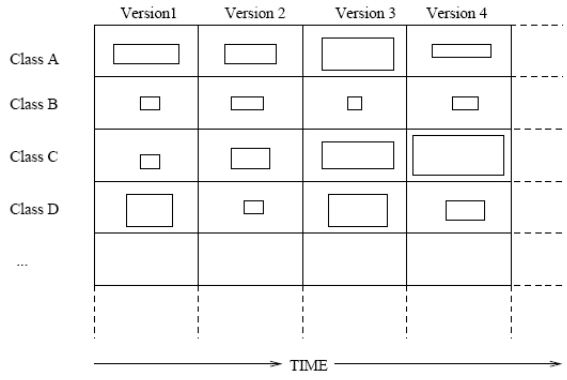


**Figure 5.** A schematic display of the Evolution Matrix

Note that with the Evolution Matrix, the analysis of classes is performed over multiple versions of the system. In our experiments we are considering a continuous reasoning about *program changes* while the programmer is working on the implementation, and we are not using specific metrics to calculate the values for classifying the classes. Our approach is used as a means to express rules that identify such classes of interest based on the history of changes related to the *number of methods* in a class.

***Pulsar***

A *Pulsar* class is a class that grows and shrinks continuously during its lifetime. The growing stage indicates an increase of functionality and the shrinking stage may indicate possible refactorings or a redesign. Note that a refactoring may also make a class grow (e.g. when a long method is divided in shorter ones). A Pulsar class reflects an increase or decrease in size in each new version of the implementation. Figure 6 graphically depicts the evolution of a *Pulsar* class upon evolution.

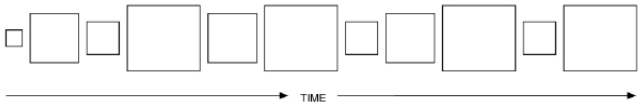


**Figure 6.** The visualization of a *Pulsar* class

We detect a possible *Pulsar* class by comparing the numbers of methods in three points in time.

```
1 pulsar(?class) if
2     pulsar(?class, ?len3, ?len2, ?len1)
3
4 pulsar(?class, ?len3, ?len2, ?len1) if
5     allConsecutiveSizes(?class,
6       ?len3, ?len2, ?len1),
7     escape(?val, [ (?len1 * 2) = ?len2
8       and:[ ?len1 = ?len3 ]])
```

The first rule **pulsar** is the general entry point of the *Pulsar* pattern and invokes the second rule. The second rule **pulsar** returns all *Pulsar* classes together with three lengths of the classes (indicating the growing and shrinking of the class). This rule uses the temporal predicate (*allpast*) by calling the predicate *allConsecutiveSizes* to retrieve the length of the class (i.e. the number of methods) at three different points in time.

Our basic relation to establish a *Pulsar* class is defined in the Smalltalk block assigned to the *escape* predicate. We consider a class as possible pulsar if the initial size (*?len1*) of the class got doubled over time (*?len2*) and afterwards fell back to its original size (*?len3*). Note that this is only a possible interpretation of the Pulsar pattern. By interchanging the Smalltalk block that compares the sizes of the class at different points in time, other definitions can be used in this rule.

This example also illustrates the symbiosis that we offer from within PARACHUT with the underlying Smalltalk

system. Remember from above that within SOUL, a developer can use Smalltalk blocks within the logic program. PARACHUT offers a similar feature by means of the *escape* predicate.

***Supernova***
A *Supernova* class is a class that from a certain point in time demonstrates a large growth. It can become a *Red Giant* class (i.e. a class bigger than a *Supernova* class and which is growing all the time). Some reasons of such a growth can be: the application of refactorings that move functionality to this class, adding functionality to retrieve data from data storage classes, and so on. These classes may indicate an unclean design or a possible introduction of bugs. The graphical representation of this pattern is shown in Figure 7.

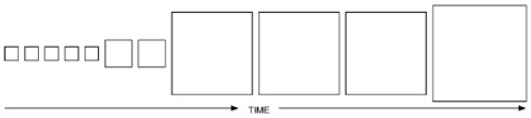


**Figure 7.** The visualization of a *Supernova* class

Similar to our rules for detecting possible Pulsar classes, we detect a *Supernova* class by comparing the numbers of methods at three points at time.

```
1  supernova(?class) if
2     supernova(?class, ?len3, ?len2, ?len1)
3
4  supernova(?class, ?len3, ?len2, ?len1) if
5     increasedSize(?class),
6     mostrecent(
7       canBecomeSupernova(?class)),
8     allConsecutiveSizes(?class, ?len3,
9           ?len2, ?len1),
10    escape(?val, [ ?len3 = (?len1 * 3)
11       and:[ ?len2 = (?len1 * 2) ]])
12
13 canBecomeSupernova(?class) if
14    numberOfMethods(?class, 20)
```

The first rule **supernova** is the general entry point of the *Supernova* pattern and invokes the second rule. The second rule **supernova** returns all *Supernova* classes together with the three lengths of the classes in which they are detected as *Supernova*. The first condition (line 5) of this rule (*increasedSize*) verifies that the class size is increasing. The second condition verifies that in the most recent past it reached the minimum number of methods to be considered a possible *Supernova* class (*canBecomeSupernova*), and afterwards all the combinations of consecutive sizes are evaluated to verify if the relation among the three lengths hold. In other words, we verify that over the course of time, the size of the class doubled and tripled, indicating the explosive growth.

The third rule **canBecomeSupernova** specifies the minimum threshold for which we possibly consider a class a supernova. For now this value is set to 20, but this can again be interchanged with an actual value depending on the system to which the rule is applied.

### 4.2 Indicators of Possible Problems

In this section, we present two examples to illustrate how PARACHUT can be used in order to detect possible problems in an implementation based on *program changes*. These experiments also show how the use of temporal predicates allow us to express elegant rules.

***Hash - Equals Rule***
We demonstrate how we can use PARACHUT to verify a typical Smalltalk rule of thumb. In order to ensure the correct comparison of objects that are part of a collection, classes in Smalltalk that implement the `hash` message should also implement the = method, and vice versa. For a developer, it might be interesting to immediately get feedback when coding about this rule of thumb. E.g. if developers implement the = message on a particular class, getting feedback reminding them to also implement the `hash` method can be quite useful.

We emphasize that is also possible to express this example using regular SOUL, however the main difference with our representation is that regular SOUL is not pro-active (i.e. does not react at the moment when the developer changes the program).

```
1  classNeedsHash(?class) if
2     methodInClass(?class, =),
3     not(methodInClass(?class, hash))
4
5  classNeedsEquals(?class) if
6     methodInClass(?class, hash),
7     not(methodInClass(?class, =))
```

The rule **classNeedsHash** detects classes that have implemented = and that have not implemented *hash*. This is done by its two conditions (lines 2 - 3), respectively. Note that in the moment a developer added the = method in class A, he immediately will know that A also needs the *hash* method, and such a warning will be reported until the *hash* method is implemented.

The rule **classNeedsEquals** is the complementary rule to **classNeedsHash**. It detects classes that have implemented *hash* and that have not implemented =, by means of its two conditions (lines 6 - 7).

In both rules the predicate **methodInClass** detects when a method is implemented (i.e. exists) in a particular class.

***Detecting possible refactoring opportunities***
This example proposes two rules that detect a possible refactoring opportunity in the source code. If there are two methods with the same selector that belong to two sibling classes, and if their parent class does not implement that common selector, then this might indicate a possible refactoring opportunity. For example, a developer might, depending on whether the two methods are identical, whether there are

other siblings implementing the method, and so on, decides to introduce an abstract method, introduce an intermediate class, or performs a push up refactoring.

```
1  duplicatedSelector(?classA,?classB,?method) if
2     addedMethodInClass(?classB,?method),
3     mostrecent(
4       addedMethodInClass(?classA,?method))
5
6  refactoringOpportunity(?superclass,?method) if
7     duplicatedSelector(?classA,?classB,?method),
8     subclassOf(?classA,?superclass),
9     subclassOf(?classB,?superclass),
10    not(addedMethodInClass(?superclass,?method))
```

To detect such possible refactoring opportunities, we define two rules.

The rule **duplicatedSelector** allows us to detect when a developer implements a method with the same name (i.e. selector *?method*) in two classes. This rule matches all changes in the system in which the developer adds a new method (using the *addedMethodInClass* predicate on line 2) named *?method* to a particular class *?classB*. A selector is considered a duplicate selector if before the addition of that method to class *?classB*, a method with the same name is also added to a class *?classA*.

The rule **refactoringOpportunity** detects possible refactoring candidates where there exists a common method *?method* in two siblings, for which the parent class *?superclass* does not provide an implementation. As a first condition of this rule, the `duplicatedSelector` auxiliary predicate we defined above will be used (line 7) to verify that there are two classes *?classA* and *?classB* that implement the method with name *?method*. If these classes have the same super class *?superclass* (lines 8 and 9), and if this superclass does not implement the method *?method*, then this situation is considered to be a refactoring opportunity.

Since we are using a forward chainer, a developer will become aware while coding of such possible refactoring opportunities. If the developer performs a refactoring, this yet again will percolate throughout the Rete network which will invalidate the above rule, resulting in that the refactoring opportunity no longer is reported.

# 5. Related Work

## 5.1 Off-line analysis of source code repositories

A first group of related tools all use information available in *traditional versioning system* such as CVS, SVN, and so on, to perform an *off-line analysis* by using a wide range of techniques (e.g. heuristics, metrics, algorithms, graphs). In comparison to our approach, these tools allow only for an *a posteriori* analysis of the changes. Furthermore, since they work on snapshots of the system, the obtained changes that are processed by these tools are generally coarse-grained.

Although a multitude of such approaches exists, we only list a few in this section. For example there is the work of Ying [21], where change patterns are used to analyze sets of files that changed together in the past. DynaMine [12] is a tool that allows for finding common error patterns from the information available in a source code repository. Hassan proposes the Top Ten List [7], a tool that highlights the ten subsystems in a system that are most susceptible to containing faults in the near future based on the current states of the project. Weissgerber and Diehl propose a technique to detect and rank refactoring candidates from software archives [18].

## 5.2 On-line analysis of changes

A second group of tools use the information concerning changes that is made available by *integrated development environments* (IDEs), such as Eclipse, Pharo, VisualWorks, etc. to monitor developers' activity in order to perform an *on-line analysis* of the system. These tools use a first-class representation of the changes that are gathered directly from the IDE in order to support software developers. Within the Smalltalk community, we can discern two such approaches. First, the SpyWare toolset by Robbes et al. [16] provides a set of tools that use a first-class representation of changes to support multiple program comprehension and reverse engineering tasks. Second, ChEOPS [4] by Ebraert et al. uses a first-class representation of changes as a means to record the implementation of features in the software and to support the composition of such features and the verification of other possible compositions.

These approaches are complementary to the work we have presented in this paper. As the underlying tool to record the changes that are entered in our Rete network, we have used the SmallBrother tool. Although this tool is able to notify us of any changes in the source code of a system, the degree of information that is provided by SmallBrother is rather restricted. It would be interesting to experiment with the change models that are proposed by SpyWare and CheOPS to see what additional information about changes, such as for example the annotations of changes that are supported by CheOPS, could be made available in the query language.

## 5.3 Time Warp

In earlier work we have introduced Time Warp [17]. Time Warp is an extension of the SOUL language that offers a means to query the history of a software system. To this end, it leverages the information that is available in the FAMIX meta-model for object-oriented programs and the HISMO meta-model that represents the history of a software system. Similar to PARACHUT, Time Warp also incorporates the use of temporal logic in order to reason about the notion of time that is inherent to the HISMO model. However, since Time Warp is an extension of the SOUL language, it suffers from the same problems with regard to supporting pro-active software engineering tools. Furthermore, similar to the off-line analysis tools we discussed above, the HISMO model on which Time Warp is based offers a coarse-grained

representation of the changes, in contrast to the fine-grained change model used by PARACHUT.

## 6. Conclusions and future work

In this paper we have presented PARACHUT. PARACHUT is a novel temporal logic query language that originated from the research of declarative meta programming at the Vrije Universiteit Brussel. In contrast to the existing SOUL language for reasoning over object-oriented programs, PARACHUT provides the following two innovations. First, by using a forward chaining inference engine, PARACHUT is able to support pro-active software engineering tools. Rather than having to manually trigger queries such as is the case with SOUL, changes in the source code of an analyzed system will percolate throughout the logic inference engine and novel results will immediately be shown to users of the tool. Second, since PARACHUT has access to a fine-grained change model of the software, it offers facilities to exploit this change model by offering temporal logic operators to reason about the temporal relations between changes.

Although we have illustrated PARACHUT by means of a couple of examples, these only scratch the surface of the expressive capabilities of PARACHUT. As an avenue of future work, we therefore propose to further investigate libraries of predicates that encode interesting knowledge that can be presented to a developer while coding. For example, using PARACHUT we could provide predicates that detect bad smells, common errors, and so on and that alert a developer pro-actively when such a bad smell or error is detected.

Furthermore — as we already discussed above — tools such as SpyWare and CheOPS provide interesting change models that appear to be richer than the change model we obtained by using SmallBrother. It would be interesting to incorporate these change models to investigate their impact on the capabilities of PARACHUT.

Finally, in this paper we have not discussed any possible scalability issues of our approach. However, when applied to large systems, the use of a Rete network can result in a large memory footprint. As such, we expect that further research is necessary in order to provide an implementation of the Rete that does not keep all the facts in memory, but e.g. uses a relational database or other persistence mechanism.

## Acknowledgments

Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This research is supported by the IAP Programme of the Belgian State.

## References

[1] C. De Roover, J. Brichau, C. Noguera, T. D'Hondt, and L. Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *ACM-SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM07)*, pages 92–102, 2007.

[2] D. Deridder. *A Concept-Centric Environment for Software Evolution in an Agile Contextards an Open Adaptive Development Environment*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2006.

[3] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *International symposium on Software Architectures and Component Technology*. Kluwer Academic Publishers, January 2000.

[4] P. Ebraert, J. Vallejos, P. Constanza, E. Van Paesschen, and T. D'Hondt. Change-oriented software engineering. In *Proceedings of the international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 3–24, 2007.

[5] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. In *Proc. 18th IEEE Symp. on Foundations of Computer Science*, page 324341, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[6] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Aspect-Oriented Software Development (AOSD)*, pages 60–69, 2003.

[7] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st International Conference on Software Maintenance*, September 2005.

[8] C. Herzeel, K. Gybels, P. Costanza, C. De Roover, and T. D'Hondt. Forward chaining in halo: An implementation strategy for history-based logic pointcuts. *Elsevier Journal of Computer Languages, Systems & Structures*, 2008.

[9] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution*, page 3742, 2001.

[10] M. Lanza. Codecrawler - lessons learned in building a software visualization tool. In *In: Proceedings of CSMR 2003, New York*, pages 409–418. IEEE Press, 2003.

[11] M. Lanza. *Object-Oriented Reverse Engineering: Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, Switzerland, May 2003.

[12] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, September 2005.

[13] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, October 2000.

[14] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views: A case study. *Elsevier Journal on Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.

[15] N. Rescher and A. Urquhart. *Temporal Logic*. Springer-Verlag, 1971.

[16] R. Robbes. Mining a change-based software repository. In *Procedings of the Fourth International Workshop on Mining*

*Software Repositories ICSE Workshops MSR '07*, pages 15–15, 2007.

[17] V. Uquillas, A. Kellens, J. Brichau, and T. D'Hondt. Time warp, an approach for reasoning over system histories. In *Proceedings of the joint International Workshop on Principles of Software Evolution - Ercim Evolution workshop (IWPSE-EVOL) (to appear)*, 2009.

[18] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. *21st IEEE International Conference on Automated Software Engineering (ASE'06)*, 2006.

[19] R. Wuyts. *A Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.

[20] R. Wuyts and S. Ducasse. Unanticipated integration of development tools using the classification model. *Computer Languages, Systems & Structures*, 30(1-2):63–77, 2004.

[21] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, September 2004.