Time Warp, an Approach for Reasoning over System Histories

Verónica Uquillas Gómez
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
vuquilla@vub.ac.be

Andy Kellens
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
akellens@vub.ac.be

Johan Brichau
Université catholique de Louvain
Place Sainte Barbe 2
B-1348 Louvain-la-Neuve
johan.brichau@uclouvain.be

Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
B-1050 Brussels
tjdhondt@vub.ac.be

ABSTRACT

The version history of a software system contains a wealth of information that can assist developers in their daily implementation and maintenance tasks. By reasoning over the role of certain code entities in previous versions of the system, developers can better understand their current state, assess the required maintenance and avoid making the same mistakes over and over again. Unfortunately, current approaches do not offer a means to easily extract specific information about the source code from such a version history. In this paper we present Time Warp, a library of logic predicates that builds on the SOUL language and the FAMIX and Hismo meta-models and that allows writing queries about the history of a system. By means of a number of concrete examples, we demonstrate how our approach can be used to express interesting queries over the version history of a system.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

Keywords

Program Querying, Source-code history

1. INTRODUCTION

Various tools have been created both in academia and industry that, by extracting information from the source code of a system, support developers in their everyday tasks such as debugging, writing tests, or assessing the impact of changes on the underlying design of a system.

One particular source of useful information is the *history* of a software system. During the development of a system, it is common practice to commit subsequent versions of that system into a versioning system. Consequently, such versioning systems contain a wealth of information concerning the implementation and the evolution of the system which can be leveraged by various tools. There exists a vast amount of work on the mining of software repositories that largely focusses on the extraction of interesting trends and patterns from the version history that is contained within a versioning repository using data mining and statistical techniques. One example of such approaches is the work of Hassan [14], who applies complexity metrics to the changes in a system in order to predict faults in that system. Another example is Zimmerman et al. [1], who by analysing the history of bug reports of a system were able to empirically demonstrate that duplicated reports provide valuable feedback to developers. While mining the repository of a software system is able to provide developers interesting feedback on their systems, such approaches are ill-suited to address *specific, custom* enquiries that a developer wants to make about the history of the source code of a system, such as "Find me all classes that at one point in the history of the system referenced classes related to logging functionality".

Source-code query languages such as SOUL [26], JQuery [18] and CodeQuest [13] have been proposed as a means to write custom queries that extract information from the source code of a system. Such query languages offer developers a set of language constructs in order to express custom queries in a declarative way. While these approaches provide a relatively straight-forward way for developers to implement queries, they only allow for reasoning about a single snapshot, i.e. a single version of a system.

In this paper, we propose Time Warp, a research prototype that allows developers to query the history of the source code of a system. Time Warp is implemented as a dedicated library of logic predicates for SOUL [26], the Smalltalk Open Unification Language, which is a PROLOG implementation in Smalltalk that in the past has been proposed as a query language for reasoning about Smalltalk, Java, C(++) and Cobol programs. Time Warp builds upon this work by making it possible to write queries that take the available history of a software system into account. Since we are reasoning about multiple versions of a system it is necessary to integrate the notion of *time* into the reasoning process (e.g. when we want to write a query that retrieves all classes that have been present in all versions of the system except for the last). We achieve this by integrating temporal operations — similar to those found in temporal logic programming — into our library of logic predicates. To represent an object-oriented program, and for representing multiple versions of such a program, Time Warp respectively leverages the FAMIX [26] and the Hismo [10] meta-models.

This paper is structured as follows. In Section 2 we present the FAMIX and Hismo models on top of which our tool is built. Section 3 introduces logic program querying (i.e. the use of a logic programming language to reason about software) and the SOUL language. Afterwards, we describe Time Warp and the library of logic predicates for reasoning about multiple versions of a system in Section 4. We illustrate the use of our approach in Section 5 by explaining three example queries for retrieving interesting information from the history of a system. We provide some avenues of future research in Section 7 after discussing related work in Section 6.

The contributions of this paper are: (1) the proposal of the use of dedicated querying facilities to reason over the history of a system and (2) the definition and implementation of Time Warp, a library of logic predicates for reasoning over FAMIX and Hismo models.

2. FAMIX AND HISMO

To represent object-oriented programs and the history of such programs, our approach leverages the work concerning respectively FAMIX [26] and Hismo [10], two models which have been developed in the context of the MOOSE analysis platform [24].

FAMIX [5] is a language independent meta-model that provides a generic representation of the static structure of programs written in several object-oriented programming languages (such as Smalltalk, Java and Python). The FAMIX meta-model consists of a set of classes that represent source code at the program entity level. Such classes are a mapping of the different elements in a program (e.g. classes, methods, attributes, comments), and of the associations between these elements (inheritance definitions, invocations of methods, and accesses to attributes by methods). Figure 1 shows the core of the FAMIX meta-model. While the meta-model is fairly complete, it can be easily extended in order to incorporate other language extensions. In Time Warp, we make use of the FAMIX meta-model in order to represent the object-oriented programs we are querying.

Since it is our goal to reason about the history of programs, we also need a representation of said history. For Time Warp, we use the Hismo model as such a representation. Hismo [10, 9] is a `transformation` of a structural meta-model into a history-aware meta-model for modelling the history of object-oriented languages. Key to this model

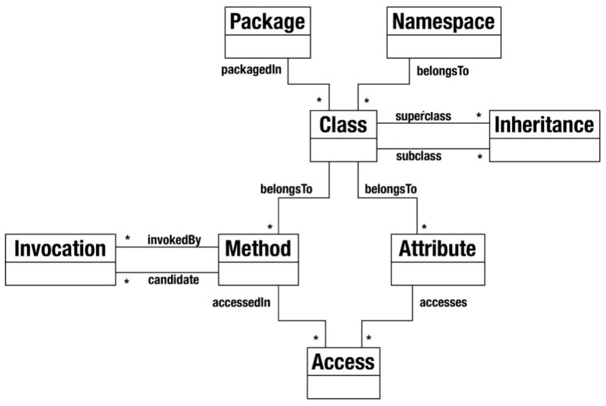

Figure 1: FAMIX core

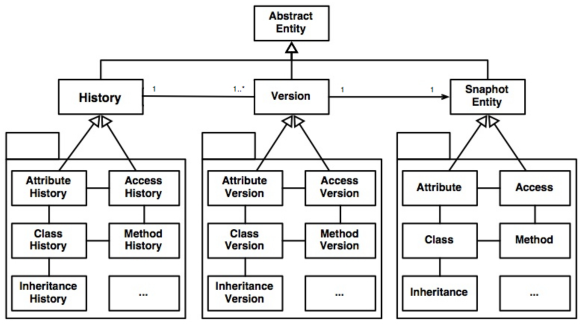

Figure 2: Part of the Hismo meta-model

is that the history is a first-class entity that can be accessed and manipulated by other tools. For example, this has been used in [11] to assess the evolution of class hierarchies.

Similar to FAMIX, Hismo uses a class hierarchy to model the different entities and relations in the version history. It offers a means to represent the versions of a subset of the entities that can be found in the FAMIX meta-model. More precisely, it models the history of packages, namespaces, classes, methods and attributes, along with the inheritance associations that are defined between the classes in the FAMIX meta-model.

Figure 2 (adopted from [10]) illustrates part of the Hismo meta-model. In a Hismo model, the history of each entity (e.g. class, method, inheritance relationship) is represented by a single history object, that contains a representation of all the versions of this entity. For example, in the figure we can see a class `ClassHistory` that is a representation of the history of one particular class. Within such a class history, each version of the class is represented by a `Version` object (for the version of a class, this object is the `ClassVersion`). Finally, each of these version objects contains a reference to the actual FAMIX entity it represents.

Note that the Hismo model is dual to the FAMIX meta-model. In other words, if there exists a relationship between two entities (e.g. an inheritance relationship between two classes), this also implies that there will be a relationship between the versions of these classes in the Hismo model (see Figure 3). Hismo models are constructed by transforming

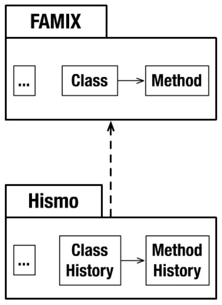

Figure 3: Transforming a FAMIX model into a Hismo model

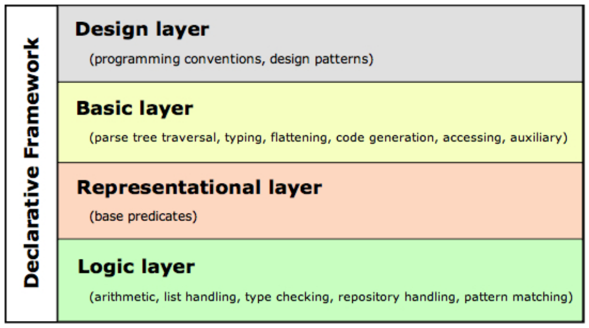
Declarative Framework
Design layer
(programming conventions, design patterns)
Basic layer
(parse tree traversal, typing, flattening, code generation, accessing, auxiliary)
Representational layer
(base predicates)
Logic layer
(arithmetic, list handling, type checking, repository handling, pattern matching)

Figure 4: The layers of SOUL predicates

snapshot meta-models, in our case FAMIX models, where each FAMIX model represents a single version of the system.

3. THE SOUL PROGRAM QUERY LANGUAGE

Time Warp is devised as a dedicated library for the Smalltalk Open Unification Language (SOUL). SOUL is a PROLOG implementation in Smalltalk for reasoning about the structure of software systems. This particular use of a logic programming language has been dubbed logic program querying. Before explaining our approach, we briefly discuss logic program querying and the SOUL language.

Logic program querying is the use of a logic programming language to write *meta-programs*, i.e. programs that reason about other programs. In other words, logic programs are used to manipulate and reason about programs written in some underlying programming language.

Logic programming languages are very suitable for writing *meta-programs* that reason over software [6, 20] because they focus on *what* the base language does by means of entities and their relations, instead of *how* the computations are executed. The logic program describes knowledge about programs written in the same or another language. In previous research, logic meta-programming has successfully been applied in software engineering to support problems such as the co-evolution of design and implementation [6, 20], or as a basis for aspect-oriented programming languages [12].

SOUL [26] is a logic program querying language designed for the reasoning of object-oriented systems, independent of their implementation language. SOUL is written in, and exists in symbiosis with Smalltalk. Therefore, it creates a symbiosis between the declarative and object-oriented paradigm. While the SOUL language is very similar to PROLOG, it provides a number of specialised features (such as linguistic symbiosis) that facilitate reasoning over software systems, as well a set of logic libraries that offer dedicated predicates for reasoning about programs written in Smalltalk, Java, C(++) and Cobol.

These libraries of predicates are divided over a number of predicate layers, as depicted in Figure 4.

Each layer contains a set of predicates oriented to specific reasoning tasks. The **logic layer** contains predicates that implement basic logic functionality (e.g. *equals*, *findall*, *number*). The **representational layer** reifies concepts of the base language in order to reason about them (e.g. *class*, *instanceVariable*, *superclassOf*, *methodInPackage*). The **basic layer** adds auxiliary predicates that facilitate reasoning about the implementation, and raises the level of abstraction significantly (e.g. *abstractClass*, *methodCallsMethod*, *instanceVariableWithName*). The **design layer** provides a number of predicates that can be used to detect bad smells, design patterns (e.g. *compositePattern*), and so on.

We illustrate the use and syntax of SOUL by means of a small example, namely the identification of accessor methods in Smalltalk programs. Below shows a prototypical implementation of an accessor method for an instance variable (field) named `amount`.

```
amount
   ^amount
```

Generally in Smalltalk, accessors are implemented by a method whose name corresponds to the instance variable that is being accessed and that consists of a single statement returning (indicated by the caret symbol) the value of the variable. Figure 5 presents the SOUL rule that we can write in order to retrieve all methods that follow this convention. Note that this query makes use of the logic library to reason about Smalltalk programs.

```
1  accessor(?class,?method,?varName) if
2   class(?class),
3   instanceVariableInClass(?varName,?class),
4   methodWithNameInClass(?method,?varName,?class),
5   methodWithReturnStatement(
6                  ?method,variable(?varName))
```

Figure 5: Query for retrieving the accessor methods

SOUL's syntax only differs slightly from PROLOG, namely that SOUL uses the keyword `if` instead of `:-` and logic variables are identified by a question mark (e.g. `?class`) instead of being capitalized. The rule in Figure 5 has three logic variables, representing the class `?class` of an accessor method `?method` along with the name of the variable `?varName` that is being accessed.

The body of the rule (lines 2 to 6) consists of four conditions that make use of the library of predicates that is offered by SOUL. An entity is considered to be an accessor method if there exists a class `?class` in the system (line 2) that has an instance variable (line 3) `?varName`. Furthermore, line 4 expresses that in this same class `?class` there must also be a method `?method` with the same name as the instance variable (`?varName`). Finally, in lines 5 and 6 we require that this method ?method has a return statement that returns the variable ?varName.

We already mentioned that SOUL is tightly integrated with the underlying Smalltalk language, resulting in a symbiosis between both languages. This symbiosis makes it possible to include Smalltalk code into SOUL queries, either as a logic condition or as a value that will be used in the computation. For example, the logic predicate class that we used in the previous example is implemented using this symbiosis:

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

Smalltalk blocks included into SOUL queries are indicated by means of square brackets ([ ]). In the above example, 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.

4. Time Warp

In this section we explain Time Warp, our library of logic predicates for the SOUL language that makes it possible to reason about the history of the source code of a system. This library of logic predicates we defined makes it possible to expressively reason over FAMIX and Hismo models and offers an intuitive means to query the version history of a system. Before presenting our library of logic predicates, we first briefly discuss temporal logic programming, an extension to logic programming which takes temporal information into account into queries.

4.1 Temporal logic programming

The term *temporal logic programming* [21] has been used to describe extensions to logic programming that integrate temporal information and reasoning into the logical framework. Temporal logic abstracts the explicit handling of time, and queries are evaluated with respect to an implicit temporal context. This general definition of temporal logic fits exactly with our intention of reasoning about the history of the source code of a software system.

At the level of the programming language, temporal logic programming adds a number of concepts (meta-predicates) to the language that can be used to reason about the underlying time model. For example, temporal relationships between facts such as **current**, **past**, **previous**, **later**, **next** and so on can be used within logic rules in order to reason over the implicit underlying time model.

In Time Warp, we apply these ideas from temporal logic programming to reason over the temporal relationships between entities in Hismo models. As we discuss later, we do this by implementing a set of dedicated logic predicates that provide abstractions similar to the ones offered by temporal logic programming to reason over the time model that is encoded in the version history as represented by Hismo models.

4.2 Library of predicates

Time Warp offers a developer a library of logic predicates that can be used to reason about FAMIX and Hismo models, while taking the notion of time that is imposed by the version history of a system into account. Similar to the design of the libraries of predicates of SOUL for reasoning about a single version of a system, our logic library follows a layered design. Figure 6 shows the five layers that are present in our system. An overview of a subset of the predicates in our library can be found in Table 1.

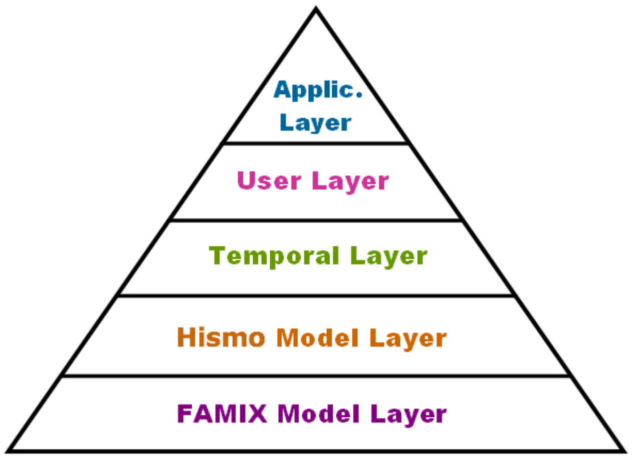

Figure 6: Library of predicates of Time Warp

4.2.1 FAMIX Model layer

At the bottom of our layered architecture, we implemented a layer of predicates for reasoning about FAMIX models. For the different entities that are present in the FAMIX meta-model, this layer contains a set of predicates that can be used to query the corresponding FAMIX entity, as well as the relationships that are present between the various entities.

For example, this layer contains predicates for reasoning over classes such as **isClass** and **classWithName**, methods (**methodInClass**, **isAbstractMethod**), fields (**attributeInClass**, **isPublicAttribute**) and so on. Aside from these predicates, this layer also contains predicates for querying basic relationships between these FAMIX entities such as inheritance relations (**isSuperclassOf**) and invocations (**methodInvokedByMethod**).

Note that since the FAMIX meta model is implemented in Smalltalk, many of the above mentioned predicates have been implemented by means of the symbiosis that SOUL offers with the underlying Smalltalk language. For example, the predicate **isSuperclassOf** is defined as:

```
1  isSuperclassOf(?superclass, ?subclass)  if
2   isClass(?superclass),
3   isClass(?subclass),
4   [?subclass inheritsFrom: ?superclass]
```

In line 4 of this rule, the inheritance link between the two classes is verified by explicitly invoking the inheritsFrom: method that is defined by the FAMIX model. In other words, this layer adds a declarative layer on top of the FAMIX meta-model that makes use of the various abstractions that are offered by this meta-model.

4.2.2 Hismo Model layer

In this second layer we provide predicates to reason about Hismo meta-models. Similar to the FAMIX layer, this layer also presents a set of predicates that align with the features

Application layer	
latestUseOfAttribute(?attribute,?history,?version)	Finds the last version in which an unused attribute was still used
frequentlyChangedMethod(?method, ?history, ?threshold)	Finds methods that were changed more than `?threshold` times
classChangedBetweenFirstAndCurrentVersion(?class, ?history)	Finds classes that were changed in between of the first and current version
polymorphicMethodAddedAfterAbstractMethod(?method, ?history)	Finds polymorphic methods for which the abstract method was added after the polymorphic method
User layer	
methodIsInvoked(?method)	Is FAMIX method `?method` invoked in one version of the system
attributeIsAccessed(?attribute)	Verifies if an attribute `?attribute` was used in the code
methodWasChanged(?version1, ?version2)	Was a given method changed in between two versions?
Temporal Layer	
previousVersionOfEntity(?previous, ?entity)	Binds `?previous` to *the* previous version of `?entity`
pastVersionOfEntity(?past, ?entity)	Binds `?past` to *a* previous version of `?entity`
nextVersionOfEntity(?next, ?version)	Binds `?next` to *the* next version of `?entity`
laterVersionOfEntity(?later, ?entity)	Binds `?later` to *a* next version of `?entity`
firstVersionOfEntity(?first, ?entity)	Binds `?first` to the first version of `?entity`
anyVersionOfEntity(?version, ?entity)	Finds any version of a particular FAMIX entity
Hismo Layer	
isHistory(?history)	Retrieves/verifies if `?history` is a Hismo history model
isClassHistory(?history)	Verifies whether `?history` is a class history model
methodVersionInSystemVersion(?methodVersion, ?systemVersion)	Extracts the method versions from a system version
systemVersionInHistory(?version, ?history)	Binds `?version` to a system version in history `?history`
methodInSystemVersion(?method, ?version)	Binds `?method` to FAMIX methods in a system version
methodOfMethodVersion(?method, ?version)	Extracts the FAMIX method from a method version
systemVersionOfEntity(?version, ?entity)	Binds `?version` to the system version in which `?entity` occurs
FAMIX Layer	
isClass(?entity)	Predicate holds if `?entity` is bound to a FAMIX class
isClassWithName(?class, ?name)	Binds `?name` to the name of FAMIX class `?class`
isPublicAttribute(?attribute)	Predicate holds if the field `?attribute` is a public field
methodInvokedByMethod(?method, ?caller)	Verifies whether method `?method` is invoked by method `?caller`
methodInvokesSelector(?method, ?selector)	Verifies whether method `?method` invokes the selector `?selector`
numberOfInvocationsOfMethod(?number, ?method)	Retrieves the number of times a method is invoked in a particular FAMIX model
isSuperclassOf(?super, ?subclass)	Checks whether a super class relationship holds between `?super` and `?subclass`

Table 1: Sub-set of predicates of Time Warp classified by layers

presented in the history models. Remember from earlier that Hismo contains three major kinds of entities, namely histories, versions and snapshots of FAMIX entities. This layer contains predicates to query all these kinds of entities. A Hismo model consists of a number of histories, that either represent the history of the entire system, or of a particular entity (e.g. `ClassHistory`). To query these entities, this layer offers predicates such as `isHistory` and `isClassHistory`, as well as predicates such as `classHistoryInHistory` to extract a class history from a particular history (i.e. a Hismo model).

Contained within these histories are the various versions of a particular entity that belongs to the system. For example, a `ClassHistory` contains a number of `ClassVersion` objects that represent each of the versions of one particular class in the system. Predicates such as `systemVersionInHistory` make it possible to query the versions of an entire system, while predicates such as `classVersionInHistory` can be used to extract the various versions of a class from a (class) history.

Finally, each version of an entity keeps an explicit link to the FAMIX entity for which it represents a version. Therefore, this layer also contains predicates that query a particular system version or entity version for the corresponding FAMIX entity. For example, for this purpose our layer offers predicates such as `classInSystemVersion` to retrieve all the FAMIX classes that are present in one particular version of a software system, as well as predicates (such as `methodInMethodVersion`) that extract the FAMIX entity from a `Version` object in the Hismo meta-model. Furthermore, this layer also contains predicates to retrieve for a particular FAMIX entity the system version in which it is present (`systemVersionOfEntity`).

Similar to the FAMIX layer, most predicates in this layer are defined by using SOUL's symbiosis to directly access the Hismo models that are present in the Smalltalk image.

4.2.3 Temporal layer

The temporal reasoning layer contains predicates that reason over the temporal relationships between Hismo entities. While the predicates in the Hismo layer provide a means to reason about the various code entities, versions and histories that are present in a Hismo model, they do not offer facilities to query the temporal relationships that exist between these entities.

The predicates in the temporal reasoning layer express the following temporal relationships over and between Hismo entities:

- Querying the version directly before or directly after a particular version of an entity (`previousVersionOfEntity`, `nextVersionOfEntity`);
- Querying any older/younger version of a particular version of an entity (`pastVersionOfEntity`, `laterVersionOfEntity`);
- Querying the first/latest version in which a particular entity is present (`firstVersionOfEntity`, `lastVersionOfEntity`);
- Querying the current (most recent) version of the system, finding the first (oldest) version of the system (`currentSystemVersionInHistory`, `firstSystemVersionInHistory`).

Note that we do not extend the SOUL language into a temporal logic programming language, but rather provide a layer of predicates that are inspired by the abstractions offered by temporal logic. This set of predicates is specifically tailored towards temporal reasoning over Hismo and FAMIX models. Since Time Warp serves as an initial prototype to experiment with the use of querying the history of the source code of a system, we do not need the full expressivity of a temporal logic programming language, but were able to incorporate reasoning about time in the library of logic predicates. These predicates make the fact that we reason about the history of a software system explicit.

4.2.4 User layer

The user layer contains a set of predicates that offer additional, reusable abstractions on top of the predicates that can be found in the three layers below this layer. These predicates extract information that is not directly available from the underlying predicates, such as the `methodWasChanged` predicate that verifies whether a method was altered between two different versions of that method, or the `attributeIsAccessed` predicate that verifies whether an attribute `?attribute` is accessed in the source code.

4.2.5 Applications layer

The final layer contains specific examples of applications of Time Warp. These predicates implement various queries for retrieving points of interest to a developer such as:

- Retrieving all classes that were added after a particular version of the system;
- Finding all the methods/classes that changed between the first and the current version of the system;
- Finding entities that were created in the first version of the system and have not been altered since;
- Finding removed/renamed entities;
- Retrieving all entities that were frequently changed;
- Finding methods that are potentially deprecated and should not be called.

In the following section, we take a look at three examples of such predicates and how they were implemented using our approach.

5. EXAMPLES

In this section, we show the use of Time Warp by means of three examples that query the version history of a system for points of interest. For each of these predicates, we discuss how they were implemented by means of the predicates in our logic library. We applied these examples to a toy system in order to validate their correctness. It is not our intent to demonstrate the scalability of our approach using these examples, but rather to show the expressiveness and capabilities of Time Warp.

5.1 Finding and tracing back unused attributes

As a first example, we demonstrate how Time Warp can be used to express a number of rules for finding attributes

```
1   attributeWithoutAccesses(?attribute,?history) if
2    isHistory(?history),
3    currentSystemVersionInHistory(?version,?history),
4    attributeInSystemVersion(?attribute,?version),
5    not(attributeIsAccessed(?attribute))
6
7   latestUseOfAttribute(?attribute,?history,?version) if
8    attributeWithoutAccesses(?attribute,?history),
9    pastVersionOfEntity(?pastAttr,?attribute),
10   attributeIsAccessed(?pastAttr),
11   not(
12    and(
13     laterVersionOfEntity(?laterAttr,?pastAttr),
14     attributeIsAccessed(?laterAttr))),
15   systemVersionOfEntity(?version,?pastAttr)
```

Figure 7: Rules for identifying the latest version in which a particular attribute was used.

(fields) that are no longer being accessed in the current version of the system, and to find out the most recent version in which they were still being used. It is not uncommon during the development of a system that particular source-code entities become redundant: for example, methods that no longer get called, and so on. For a developer it is not only interesting to know about these entities, but also to get additional information about the context in which these entities became redundant. For example, if developers have information about the version in which one particular source-code entity was used for the last time, they can then further query this version of the system in order to assess the reasons why this entity was not used any more later on.

In Figure 7, we demonstrate two rules for retrieving all attributes that are no longer used in the current version of the system, along with the most recent version of the system in which the attribute *was* still used. Using the predicates offered by Time Warp, a developer can then further query that version of the system in order to find out who used the attribute.

First, we define an auxiliary predicate `attributeWithoutAccesses` (lines 1 to 5). This predicate retrieves all the unused attributes `?attribute` that are present in the system. This is achieved by asking the system history `?history` for the current version `?version` of the system (lines 2 and 3). Within this current version, we query all attributes `?attribute` (line 3) that are no longer accessed anywhere in the source code (line 4).

Lines 7 – 15 show the predicate `latestUseOfAttribute` that queries the version history for the most recent version of the system in which a currently unused variable was still used. Line 8 uses the predicate `attributeWithoutAccesses` we defined above to retrieve attributes `?attribute` that are no longer accessed. We then query the system for any past versions `?pastAttr` of this attribute (line 9), that *were* accessed in the version of the system in which they appeared (line 10). Since we are interested in finding the most recent version of the system in which the attribute was still used, we state that there should be no later version (than version `?pastAttr`) of the attribute that was still accessed (lines 11 – 14). Line 15 retrieves the system version `?version` in which the attribute `?pastAttr` was most recently used.

Note however that the above rules do not retrieve attributes that are not used in *any* versions of the system.

```
1   frequentlyChangedMethod(?method,?history,?threshold) if
2    isHistory(?history),
3    currentSystemVersionInHistory(?version,?history),
4    methodInSystemVersion(?method,?version),
5    countall(
6     ?vmethod,
7     and(anyVersionOfEntity(?vmethod,?method),
8      previousVersionOfEntity(?prevmethod,?vmethod),
9      methodWasChanged(?vmethod,?prevmethod)),
10    ?number),
11   [?number >= ?threshold]
```

Figure 8: Definition of the `frequentlyChangedMethod` rule.

5.2 Frequently changed methods

The second example demonstrates the use of our approach to implement a simple metric over the history of the source code. More specifically, we want to retrieve all frequently changed methods that are present in the current version of the system. To this end, we have implemented a rule `frequentlyChangedMethod` that is depicted in Figure 8. This predicate queries the version history for a method `?method` that is present in the current version of the system (lines 2 to 4). For this method, we count (using the `countall` higher-order predicate) the number of times `?number` that the method changed during the version history of the system (lines 5 – 10). This is achieved by querying from within the higher-order predicate any version `?vmethod` of the method `?method`, along with the version `?prevmethod` that is previous to version `?vmethod`. With the predicate `methodWasChanged` (line 9) we verify that the method was changed in between version `?vmethod` and `?prevmethod`. The method is considered to be frequently changed if the number of changes `?number` is larger than a user-specified threshold `?threshold` (line 11).

5.3 Finding possibly undesirable method invocations

As a final example we demonstrate the use of our approach to query the version history of a system for possibly undesirable invocations to particular methods. Consider the situation (see Figure 9) where we have a software system with n different versions. In previous versions of this system (e.g. $version_{n-y}$ and $version_0$) a method `m` was called. In a version $version_{n-x}$ of the system that same method `m` does not get called, but it is still present in the system. Afterwards (for example in the current version of the system $version_n$) this method `m` yet again gets called.

While the above situation does not necessarily indicate an error in the system, it can however be interesting for a developer to be warned about such methods that possibly should not have been called in $version_n$:

- For instance, a developer might have deprecated method `m` in $version_{n-x}$ of the system, but did not remove the method due to compatibility reasons. While it is considered good practice to indicate that the method became deprecated (by e.g. annotating it), the developer might have failed to do so. Any calls to this method in a later version ($version_n$) can then be considered as errors;

- Similarly, method `m` might have become dead code in

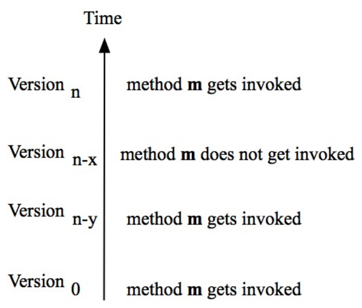

Figure 9: An illustration of possibly undesirable method invocations.

$version_{n-x}$. If this dead code gets invoked afterwards, this is a point of interest to a developer.

```
1 possiblyUndesiredInvocation(?method, ?history) if
2   isHistory(?history),
3   currentSystemVersionInHistory(?version,?history),
4   methodInSystemVersion(?method,?version),
5   methodIsInvoked(?method),
6   pastVersionOfEntity(?oldMethod,?method),
7   not(methodIsInvoked(?oldMethod)),
8   pastVersionOfEntity(?olderMethod,?oldMethod),
9   methodIsInvoked(?olderMethod)
```

Figure 10: The predicate to identify possibly undesirable method invocations.

Figure 10 shows the implementation of the predicate `possiblyUndesiredInvocation` that queries the history of a system for the kind of methods we described above. Lines 2 and 3 of the rule retrieve the current version `?version` from a history model `?history`. From this version of the system, we retrieve methods `?method` that are (possibly) invoked by at least one other method (lines 4 and 5). As described above, for this method `?method` to be possibly undesirably called, there should exist an old version `?oldMethod` of the method that, in the version which it belongs to, was not invoked (lines 6 and 7). Furthermore, for this old version `?oldMethod`, there should be an even older version `?olderMethod` that *does* get invoked (lines 8 and 9).

6. RELATED WORK

Reasoning over Hismo.
In the context of the Moose project, a number of tools have been proposed such as CodeCity [25] and Van [10] that make use of the FAMIX and HISMO meta-models. Since the Hismo meta-model is an object-oriented representation of the history of a software system, this model can directly be queried using the Smalltalk language. This however poses two downsides. First, a user who wants to query the model needs to be fully aware of the details of the entire model in order to traverse it and extract the necessary information. Our approach tries to hide the internal implementation of Hismo as much as possible by offering a declarative means to query Hismo models, rather than having to imperatively traverse a Hismo model. Second, our library of predicates introduces an explicit time model into the query language.

Logic software query languages.
The idea of using a logic programming language to query software systems is by no means novel. One direction that has been investigated in this field is the use of a Turing-complete logic programming language to query the source code of a system. For instance, the SOUL language [26] as well as the TyRuBa language which underlies the JQuery tool [4] use a PROLOG dialect to offer an expressive means to query source-code entities and the relationships between these entities.

The work of Verbaere and De Moor concerning CodeQuest and SemmleCode [13, 3] provides a different approach that favours performance over expressivity. Rather than using a Turing-complete language, these approaches respectively use Datalog [2] and QL [8], languages that only offer a subset of the PROLOG language by e.g. limiting the possible forms of recursion and excluding the definition of data structures. Using these restricted languages, they are able to provide very efficient query languages for reasoning over object-oriented systems.

To the best of our knowledge, the existing logic query languages however only support reasoning over a single version of the software system. As such, our approach contributes to this field by providing a logic library that makes it possible to query the version history of a system.

SCQL.
Hindle and German [17] propose SCQL, a dedicated formal model and query language for reasoning over source code repositories. Similar to Time Warp, their approach uses a logic query language based on first-order logic and temporal logic to reason about version control repositories. As the underlying model which they reason about, they propose a graph representation of a repository containing information about the different revisions, files and authors, and also include information such as the commit message, the timestamp of the commit and so on. In contrast, our approach does not reason about the revision history, but about the history of the actual source code entities.

Aspect-oriented programming.
Within the field of aspect-oriented programming, the HALO language [16] has been proposed as a means to express pointcuts based on temporal logic programming. Using HALO, a developer can write pointcuts that capture join points (i.e. execution-time events) based on information that is present in the history of all already executed join points. In this approach, the temporal logic primitives are used to reason about the history of join points and to write pointcut expressions in terms of this history. In contrast to our approach, HALO does not reason about the version history of the system, but rather about the history of an execution of the system.

Mining software repositories.
A vast amount of related research exists within the mining software repositories community. These approaches use the information that is available within traditional versioning systems such as CVS and Subversion in order to support several software engineering tasks. For example, data mining techniques and heuristics/metrics have been applied in this context as a means to extract valuable information from software repositories. While a complete overview of

the field of mining software repositories lies outside of the scope of this paper, we present a couple of interesting approaches. Hassan [14] proposes a technique to perform fault prediction in a system by applying complexity metrics on the changes that are present in the repository. Zimmerman et al. [1] were able to identify, by studying the history of bug reports, a positive correlation between duplicated bug reports and the benefits in terms of additional information that developers obtain by these duplicated reports. Source Sticky Notes [15] is an approach that annotates a static dependency graph of a system with useful information that is extracted from the history of a system, as a means to help developers to understand the context of the changes they are applying. DynaMine [19] is a tool that applies data mining techniques on version archives to find common usage patterns by analyzing co-changed methods.

Approaches such as the ones described above provide developers useful information by mining the software repository: they look for interesting/meaningful patterns and information in a repository in order to support a particular development task. The goal of our approach is different: we do not intend to support a particular development task but rather offer a general platform for querying the history of the system. Our approach however is limited to querying the history of the source code of the system. Often, a software repository contains other kinds of information (such as bug reports) that is currently not leveraged by our tool.

Capturing software changes.
While the approaches described above — similar to our approach — provide an off-line analysis of the development history, there are approaches such as SpyWare [22, 23] and CheOPS [7] that do an on-line analysis of the source code's history. These approaches are tightly integrated with a developer's environment and use this environment to monitor and record the changes that the developer makes to the source code. Based on these recorded changes, both approaches present a set of tools to aid a developer in tasks such as program comprehension and feature composition validation.

The main advantage of these approaches is that they offer a very detailed model of the history of the source code of a system. Each change that was made to the source code is logged in the order that it happened. As such, one can get additional information concerning e.g. the order in which various classes in one version of the system were modified/added. In principal, it is possible to use such a fine-grained history model as the underlying model for our approach. How this impacts the expressiveness and usability of our approach is a topic for further investigation.

7. FUTURE WORK AND SUMMARY

In this paper we have presented Time Warp, a tool for querying the version history of the source code of a system. Our approach is based on the FAMIX and Hismo meta-models and offers a library of dedicated predicates to reason about these meta-models, as well as to express temporal relationships between the entities in both models.

In this paper, we have demonstrated the applicability and expressiveness of our approach by means of three small examples. By no means does this serve as a complete validation of our approach. While we have applied these examples to a toy system in order to ensure the correctness of our library of predicates, this does not allow us to make any claims about the scalability of our approach when applied to large case studies. Part of our future work therefore is to perform a complete validation by means of reasoning over the version history of a large-scale system such as Eclipse.

Time Warp offers a prototype implementation to demonstrate the use of querying the version history of a system. While our implementation was conceived as a library of logic predicates, a further integration with the actual query language is possible. One avenue of future research is to extend the SOUL language itself such that it supports temporal logic programming to reason about the history of a software system. The library of logic predicates we presented in this paper is sufficiently rich to reason about the FAMIX and Hismo models. However, a full-fledged temporal program query language would make it easier to integrate other underlying time models, as well as offer a generalised declarative framework for using and composing predicates that reason about a system's history.

While not reported in the paper, the implementation of our approach required us to extend FAMIX and Hismo in a number of places, in order to e.g. keep a reference to the actual source code of methods over the entire version history. Moreover, Hismo is implemented by storing copies of the entire FAMIX models. We feel that such a scheme might limit the scalability of our approach, and are planning to investigate the incremental specification of the history model: rather than copying the entire FAMIX model in each of the versions, only keep track of the delta between the various versions.

Finally, up until now we have only focussed on reasoning over the history of the source code of the system. As has been illustrated by the work in mining software repositories, these repositories often also contain other useful information such as bug reports, comments, and so on. A possible future research track is to incorporate also these kinds of information in the logic library.

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. The authors would like to thank Kris De Schutter and Tudor Girba for their comments on earlier drafts of this paper.

8. REFERENCES

[1] N. Bettenburg, R. Premraj, T. Zimmerman, and S. Kim. Duplicated bug reports considered harmful ... really? In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 337–345, 2008.

[2] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[3] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble. Keynote address: .ql for source code analysis. In I. C. Society, editor, *Proceedings of the seventh IEEE International Working Conference on Source Code*

Analysis and Manipulation (SCAM), pages 3–16, Washington, DC, USA, 2007.

[4] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *Practical Aspects of Declarative Languages*, volume 3819 of *Proceedings of the ERCIM Working Group on Software Evolution*, pages 88–102. Lecture Notes in Computer Science, 2006.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1: The famoos information exchange model. Technical report, 2001.

[6] 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.

[7] 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.

[8] I. Futó, F. Darvas, and P. Szeredi. The application of PROLOG to the development of QA and DBM systems. In H. Gallaire and J. Minker, editors, *Logic and Databases*, New York, 1978.

[9] T. Girba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Switzerland, November 2005.

[10] T. Girba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, May 2006.

[11] T. Girba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 2–11, 2005.

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

[13] E. Hajiyev, M. Verbaere, O. de Moor, and K. De Volder. CodeQuest: Querying source code with DataLog. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 102–103, 2005.

[14] A. Hassan. Predicting faults using the complexity of code changes. In *31st International Conference on Software Engineering (ICSE)*, 2009.

[15] A. Hassan and R. Holt. Using development history sticky notes to understand software architecture. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, 2004.

[16] 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.

[17] A. Hindle and D. German. SCQL: A formal model and a query language for source control repositories. In *Mining Software Repositories*, pages 100–105, 2005.

[18] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Aspect-oriented software development*, Proceedings of the second international conference on Aspect-oriented software development (AOSD), pages 178–187, 2003.

[19] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. *SIGSOFT Software Engineering Notes*, 30(5):296–305, September 2005.

[20] 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.

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

[22] R. Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories ICSE Workshops MSR '07*, pages 15–15, 2007.

[23] R. Robbes and M. Lanza. Spyware: A change-aware development toolset. *Proceedings of the 30th International Conference on Software Engineering 2008*, pages 847–850, May 2008.

[24] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Switzerland, December 2001.

[25] R. Wettel and M. Lanza. Visualizing software systems as cities. In *4th International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT)*, pages 92–99, 2007.

[26] 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.