High-Level Polymetric Views of Condensed Run-time Information

Stéphane Ducasse
Software Composition Group
University of Bern, Switzerland
ducasse@iam.unibe.ch

Michele Lanza
Software Composition Group
University of Bern, Switzerland
lanza@iam.unibe.ch

Roland Bertuli
I3S Laboratory
Sophia-Antipolis, France
bertuli@essi.fr

Abstract

Understanding the run-time behavior of object-oriented legacy systems is a complex task due to factors such as late binding and polymorphism. Current approaches extract and use information from the complete *execution trace of a system. The sheer size and complexity of such traces make their handling, storage, and analysis difficult. Current software systems which run almost non-stop do not permit such a full analysis. In this paper we present a lightweight approach based on the extraction of a condensed amount of information, e.g., measurements, that does not require a full trace. Using this condensed information, we propose a visualization approach which allows us to identify and understand certain aspects of the objects' lifetime such as their role played in the creation of other objects and the communication architecture they support.*

Keywords: software visualization, reverse engineering, run-time information, object-oriented programming, program understanding

1. Introduction

Corbi [4] reported that during maintenance professionals spend at least half of their time analyzing software to understand it. Moreover, Sommerville [27] and Davis [5] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system. These facts show that understanding applications is one of the hardest tasks in the maintenance of software systems.

One approach is to support the understanding of object-oriented systems using dynamic information. An extensive amount of research has already been dedicated to this research field [12] [21] [31] [22] [17] [14] [25] which is mostly based on extracting information generated during the execution of a software system. To do so people instrument the source code using various techniques (method wrapping, logging, extended virtual machines) and then run the system. The instrumented source code generates an *execution trace*, which contains information about the run-time behavior of the system, *e.g.,* which method is calling which other methods, which objects are created at which time by which other objects, etc.

The problem is that often the event trace for large, complex, long-running systems is voluminous [26]. Our experience shows that collecting a trace of a couple of seconds generates tens of thousands of events. Moreover, the low-level nature of the information contained in such a trace makes it hard for a software engineer to infer higher-level information about a software system. For example, he may want to know which other objects a certain object is sending messages to, but he does not want to have to analyse and verify every single method invocation contained in the trace. Many approaches based on run-time information have therefore as primary goal the reduction of the complexity of the trace in order to reveal certain aspects like the collaboration between classes [17] [25] [3].

Our approach is based on a *condensed* set of information, *i.e.,* we do not keep and analyze the complete trace, we rather compute simple run-time metrics by *counting* certain events. For example, we count how many methods of a class are invoked during the program execution. Then we visualize this value, representing the number of the invoked methods of a class, in a *run-time polymetric view*. Polymetric views, presented by Lanza and Ducasse [19] [20], are visualisations enriched with software metrics. These views allow us to support the understanding of the run-time behaviour of a large, complex, and long-running systems. This allows us to reduce the amount of information to a few metrics, while still obtaining and visualising valuable information about the run-time behaviour of the system.

In this article we first discuss the problems and challenges that the run-time analysis of object-oriented applications poses (Section 2). We then present our approach in detail (Section 3), before presenting and analyzing a case study performed using our tool, DIVOOR, an extension of CodeCrawler [18], with different run-time polymetric views (Section 4). We conclude the paper with a discussion of our findings (Section 5), a look at the related work (Section 6), and an outlook on our future work (Section 7).

2. Run-time Analysis Problems

Wilde and Huitt [32] assessed that understanding object-oriented applications is difficult because of several reasons, such as:

- Polymorphism and late-binding make traditional tool analyzers like program slicers inadequate. Data-flow analysers are more complex to build especially in presence of dynamically typed languages.
- The use of inheritance and incremental class definitions, together with the dynamic semantics of *self* and *this*, make applications more difficult to understand.
- The domain model of the applications is spread over classes residing in different hierarchies and/or subsystems and it is difficult to pinpoint the location of a certain functionality.

Using dynamic information is one way to support the understanding process. In such a context essential questions need to be answered such as:

- What are the most instantiated classes?
- What are the classes having tenured objects? From an architectural point of view detecting singletons is also valuable information.
- What are the classes that create objects? Detecting object factories is important information.
- How do classes communicate with each other?
- Which percentage of the methods defined in a class are actually used?

2.1 Challenges and Constraints

The run-time analysis of object-oriented systems is challenging because of the following constraints:

- *Amount and density of information*. The execution traces generated for run-time analysis are packed with *very large* amounts of low-level information. Therefore they must be analysed using techniques which reduce their complexity, such as filtering, clustering, slicing, concept analysis, or visualization [28] [26] [13] [3].
- *Granularity of information.* Execution traces contain large amounts of low-level information, *e.g.,* which methods invoke which methods, which methods access which attributes, which objects are created at what time, etc. It is difficult, using such small pieces of information, to gain an understanding at a higher level.

3. Our Approach

In a run-time context numerous classes, complex inheritance and containment hierarchies, and diverse patterns of dynamic interaction all contribute to difficulties in understanding, reusing, debugging, and tuning large object-oriented systems. Many of these difficulties are due to the amount and quality of the data that can be gathered using execution traces. The relevant information we are looking for is often hidden in huge amounts of data that must first be filtered and analysed. Our approach uses a condensed set of information to support the understanding of the run-time behavior of an application.

3.1 The Principle of a Polymetric View

The baseline of our work is based upon the approach implemented in the tool CodeCrawler [18] [19] [20]. It implements the polymetric views, software visualizations enriched with software metrics, described by Lanza [19], which we mainly used to support the understanding of software systems with static analysis. In Figure 1 we see that, given two-dimensional nodes representing entities (*e.g.,* software artifacts) and edges representing relationships, we enrich these simple visualizations with up to 5 metrics on these node characteristics:

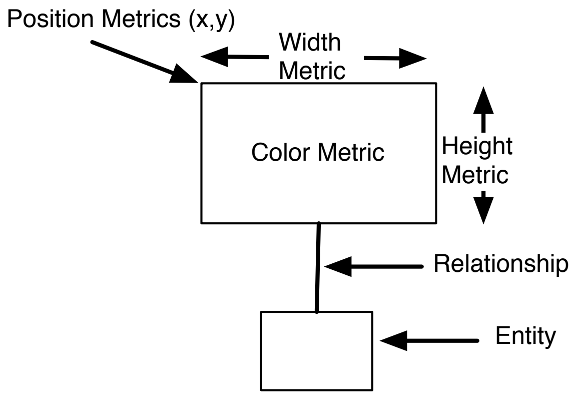

Figure 1. The principle of a polymetric view.

- *Node Size.* The width and height of a node can render two measurements. We follow the convention that the wider and the higher the node, the bigger the measurements its size is reflecting.
- *Node Color.* The color interval between white and black can display a measurement. Here the convention is that the higher the measurement the darker the

node is. Thus light gray represents a smaller metric measurement than dark gray.

- *Node Position.* The X and Y coordinates of the position of a node can reflect two other measurements. This requires the presence of an absolute origin within a fixed coordinate system, therefore not all layouts can exploit this dimension, particularly tree layouts.

In our previous work [20] we only made use of static information and software metrics which could be gathered from the static analysis of a software system. In this article we also make use of the thickness of the edges to render measurements. This gives us information about the *weight* of an edge between two entities, *e.g.,* a thick invocation edge between two classes signifies there are many invocations between the two classes.

Note that since run-time information tends to be non-linear and with huge differences in scale (for example, a method can be invoked 10 times, while another one 50'000 times), some of the views use a logarithmic scale to display the measurements.

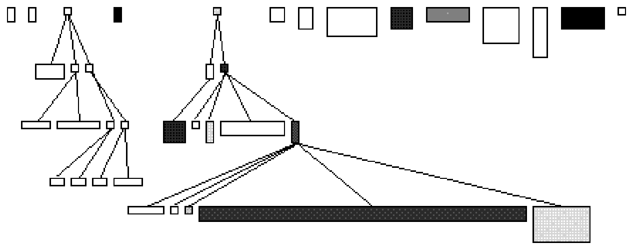

Figure 2. An inheritance tree enriched with run-time measurements.

Example. Figure 2 shows an example of an inheritance tree enhanced with run-time information. The nodes represent the classes of the analyzed application, the edges illustrate the inheritance relationships. In this example, the width of the nodes reflects the number of created instances, while the height represents the number of used methods during the execution. The color tone represents the number of method calls. For example the flat dark node at the bottom of one the hierarchies represents a class which is heavily instantiated (the node width is remarkable compared to the other nodes), but who seems, once instantiated, to perform little behaviour (the node has a small height), *i.e.,* only few methods of this class have been invoked during the analysed run-time.

3.2 Run-Time Information Collection

Run-time information collection is a rich domain that goes from the wrapping of methods [2] and the control of objects [8] to the instrumentation of virtual machines.

A great body of work has been done on run-time information in terms of a trace of events [10] [11] [24]. This trace represents valuable information but is space-consuming and complex and therefore requires additional abstractions and manipulations to extract the wanted information.

Our approach focuses on collecting a reduced amount of information, namely measurements, during the execution, such as the number of invocations, number of object creations, number of used classes/methods, the number of method calls on a class during the execution, etc.). We constrain ourselves to use only relatively simple measurements.

Class Run-Time Measurements	
Name	**Description**
NCM	Number of called methods
RCM	Ratio of called methods vs. not called methods
NMI	Number of method invocations on a class
NIMI	Number of internal method invocations on a class
NEMI	Number of external method invocations on a class
NCCM	Number of called class (static) methods
NCMI	Number of class (static) method calls on a class
NCI	Number of created instances of a class
NCO	Number of created *objects* by the class instances

Method Run-Time Measurements	
Name	**Description**
TI	Total number of calls
ITI	Number of invocations (caller is receiver of invocation)
ETI	number of invocations (caller is different than the receiver)

Table 1. A list of the measurements we extract from an execution trace.

The measurements we currently extract from an execution trace are listed in Table 1 [1].

4 Run-time Polymetric Views

In this section we define 3 run-time polymetric views and show the result of their application and analysis on a concrete case study that we present below. For the experience, we run the application on a scenario covering its different aspects.

[1]At first sight the difference between NCM and NMI can be delicate to grasp. NCM represents the number of called *methods* of a class, while NMI represents the number of *invocations* on the methods of a class. For example, if a class has 5 methods and during the execution 3 different methods have been invoked 500 times, then NCM equals to 3 while NMI equals to 500.

4.1 Case Study in a Nutshell

The particular software system used in our experiment is the Moose reengineering environment that is developed by our group [9]. Moose is a relatively small case study consisting of 137 classes and 2093 methods of Smalltalk code. It serves as a foundation for other reverse engineering tools. It provides a language independent representation and manipulation of source code written in C++, Java, Cobol, and Smalltalk. To achieve this language independence it is based on the FAMIX meta-model [7], which describes how elementary source code elements such as attributes, methods, classes, and namespaces are represented [6]. Moreover, Moose describes meta-models as instances of its own meta-meta-model. This explicit description of meta-models supports the creation of generic model reader and writers.

To parse the source code of applications of Java or C++, Moose interprets CDIF or XMI compliant files, while for extracting Smalltalk applications, Moose uses its own parser and analyzes the resulting abstract syntax trees to generate Moose models.

Run-time scenario. We run the Moose system for the analysis of a Smalltalk application: therefore a meta-model is created, a Smalltalk-specific source code model of the application is created, then the application is analyzed (metrics are computed and extracted), finally, using two external representation formats (CDIF and XMI), the model is saved on file and reloaded multiple times.

4.2 The Instance Usage Overview

Instance Usage Overview Description	
Layout	Inheritance tree, without sort
Nodes	Classes
Edges	Inheritance
Scope	Full system
Metric Scale	Logarithmic
Node Width	NCI (*Number of created instances*)
Node Height	NCM (*Number of called methods*)
Node Color	NMI (*Number of method invocations on a class*)
Figure	Figure 3

View Intention. The *Instance Usage Overview* shows which classes are instantiated and used during the system's execution. As shown by the view description above, the node width represents the number of created instances, the height of a node represents the number of methods that have been used, and the color represents the total number of method invocations during the program execution.

Revealing Symptoms. Note that this view only considers instance method invocations and does not take into account class or static method invocations. While this view provides an overview of a complete application it also offers detailed information, such as:

- Small, white, and square nodes represent classes that have not been instantiated, therefore not used.
- Narrow, pale nodes represent classes whose methods have been invoked but having no or few instances. This can be the case of singletons or abstract classes which are not instantiated but their methods are used by means of inheritance.
- Flat, pale nodes represent classes that are heavily instantiated but not often used as their number of invocation is low (denoted by the light color).
- Flat, dark nodes represent classes that are heavily instantiated with few used, but heavily invoked, methods.
- Large, dark nodes represent classes that have been heavily instantiated and whose behaviour has been greatly used.

Case Study. Figure 3 shows a part of the *Instance Usage Overview* applied on our case study.

The dark, large node annotated as *A* in our view is a CDIF scanner, which parses files written in the CDIF format, an old industrial exchange format. An instance of the scanner is created each time a model is loaded into memory. It is heavily invoked since the scanning is a dense process putting in movement many small and specific methods.

The dark, large node annotated as *B* represents the Moose meta-meta-model *AttributeDescription* class which has been instantiated a high number of times. This meta-meta-model class is instantiated to represent the current Moose meta-model. As Moose is a dynamic environment and meta-models can be extended, the current meta-model representation is created each time a model is loaded. Several methods (*i.e.,* mainly accessors) are then executed to populate the meta-model entity from a meta-meta-model description. This explains the high number of created instances. However, we were puzzled by the fact that those classes are so heavily instantiated and used (350,000 calls and 3,500 instances). Further inspections may yield chances for optimizations.

The FAMIX meta-model classes (represented in the inheritance hierarchy whose root is called *C* on the figure) which model the source code entities are flat, lightly colored nodes. Indeed the models loaded into Moose during the scenario are simple models containing only a couple of classes. Hence these classes are not the most instantiated

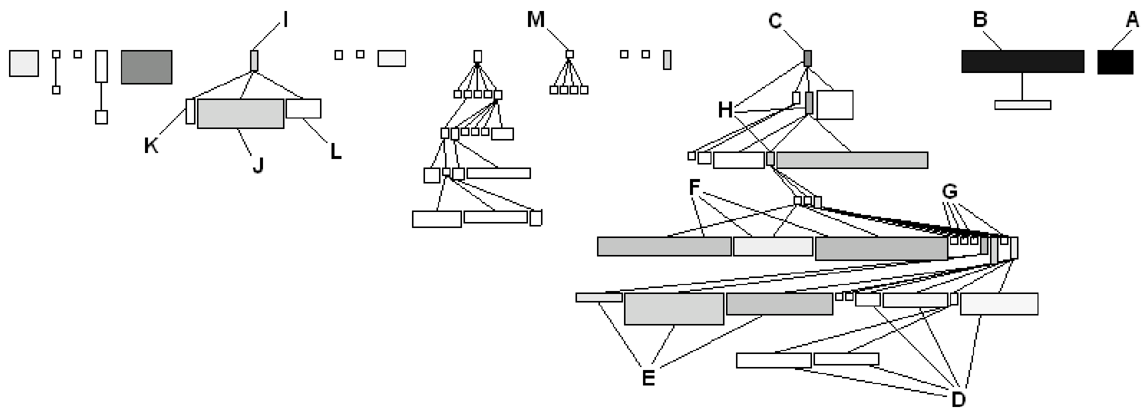

Figure 3. The Instance Usage Overview.

as would be case with the loading of large models. This inheritance hierarchy contains the three following shapes:

1. The flat nodes are the information extracted from Smalltalk code (Classes, Methods, Attributes, Inheritances, ...), and they occur always as leaves of the inheritance tree. The white classes (D) that model instance and local variables are less instantiated and used compared to the two classes modeling variable accesses and method invocations (E, F).

2. The small, square leaf nodes (G) represent classes that are defined in the language independent meta-model but that are not relevant in Smalltalk (classes modeling entities such as Includes, SourceFiles, Functions, etc.). Therefore these classes have not been instantiated at all.

3. The narrow nodes in the middle of the hierarchy (H) represent abstract classes as they are not instantiated but their methods are invoked by subclass instances.

The small hierarchy (under the class annotated I) represents the visitors parse tree that extracts the FAMIX meta model from the Smalltalk source code. The class *VWParseTreeEnumerator* [2] (J) is invoked each time a model is created from Smalltalk source code while the other two Visitors, which are *VWParsetreeMetricCalculator* (K) and *VWParseAnnotator* (L), are dedicated to analysis that is only performed on demand.

Finally the small hierarchy (M) is not covered at all by our execution. In fact, these classes represent a part the *graphical user interface* of Moose, and have not been used during the execution.

[2]VW stands for VisualWorks, a Smalltalk development environment and distribution.

Discussion. The *Instance Usage Overview* gives an overview of the run-time behavior of a whole application. It gives clues on the classes used in the system in the context of superclass code reuse. This view has the double advantage of combining static (inheritance shape of the system, number of classes) and run-time information for each class (number of created instances, number of method calls, number of invoked methods). It helps to identify *often instantiated classes*, *not instantiated classes*, *heavily used classes*, and *unused classes*.

4.3 The Communication Interaction View

Communication Interaction Description	
Layout	Embedded Spring Layout
Nodes	Classes
Edges	Invocations
Scope	Full system
Metric Scale	Linear
Node Width	NCM (*Number of called methods*)
Node Height	NCM (*Number of called methods*)
Node Color	NMI (*Number of method invocations on a class*)
Edge Width	Number of Invocations Between two Classes
Figure	Figure 4

View Intention. The *Communication Interaction* view shows the communication between classes of a system during its execution. As described above the size of a node represents the number of methods used and the color the number of method invocations. The *Communication Interaction* view takes advantage of an embedded spring layout: it weights the springs on the edges so that classes heavily communicating with each other will aggregate themselves. In addition, the width of the edges represents the class-to-class communication.

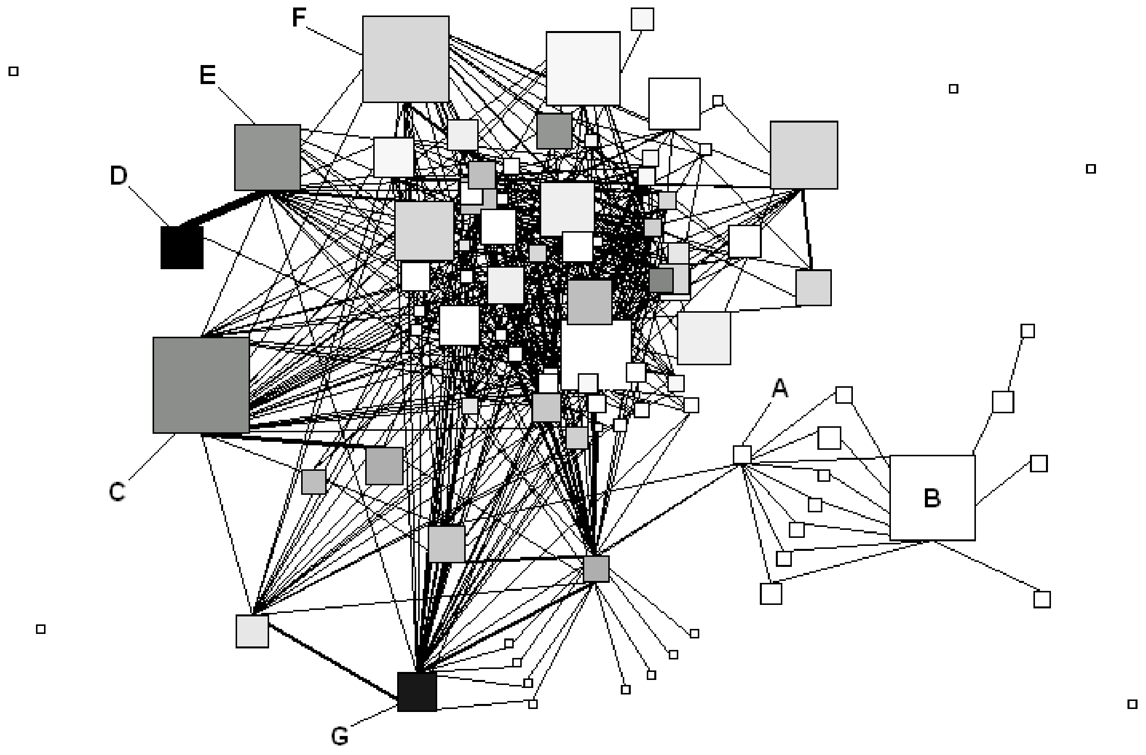

Figure 4. The Communication Interaction view.

Revealing Symptoms. This view only considers instance level method invocations and not direct class references or instance creations between classes. This view may contain:

- Unconnected, tiny, square nodes represent classes whose methods do not invoke other methods or get invoked by other methods.
- Connected, tiny, square nodes represent classes whose methods are rarely invoked. Note that such classes can still have methods heavily invoking other methods, represented as a dark class node.
- Large, white, square nodes represent classes having a considerable number of method used but which are rarely invoked during the execution.
- Dark, square nodes represent heavily used classes. Small, dark, square nodes in addition represent classes whose few methods are heavily used.
- Groups of nodes loosely connected to the view core represent classes communicating via a funnel [18] [20] to the rest of the system.

Case Study. Figure 4 shows the application of the *Communication Interaction* view on our case study.

A group of classes is clearly disconnected from the rest of the core of the view, while it joins the biggest part of the view through the class annotated as *A*. This group of classes implements the XMI file production based on a MOF compliant interface and is disconnected from the rest because the XMI/MOF production is an independent package used by Moose to produce XMI model files. The big class (B) is the XMI producer which uses MOF interface objects that communicate via a bridge class (A) with the FAMIX-compliant meta-model. The XMI producer is rarely used because Moose favors the CDIF exchange format. This explains why its class node is pale.

The big class (C) is the central repository storing all the analyzed models. Moreover it acts as a main entry for querying the models. That is why this class is connected to all the classes modelling the Smalltalk source code. The medium sized dark class node (D) is the CDIF scanner that is mainly invoked by an importer class (E) which loads models into memory. The importer has the responsibility to populate a model and as such to transform textual representations (from a CDIF text file) into objects. Note that the Moose developers learned that this class was also invoked by another one as shown in Figure 4. The big class (F) represents the class *MSEClass* that models classes in Moose. The class (G) is the class representing the meta-

meta-model of Moose which is then instantiated to represent the FAMIX meta-model. This class is used by all the FAMIX classes as they describe themselves automatically and by the input/output tools as they provide meta-model independent functionality.

Discussion. The *Communication Interaction* view identifies heavily invoked classes, but it is less scalable than *Instance Usage Overview* because when the classes communicate heavily a naive embedded spring layout has difficulties to create well identifiable groups of classes. Note also that in our approach we took into account as invocations self sends between classes and subclasses which make the underlying view much denser. Another way to reduce such a high degree of coupling would be to group all the classes within a common hierarchy.

4.4 The Creation Interaction View

Creation Interaction Description	
Layout	Embedded Spring Layout
Nodes	Classes
Edges	Instantiation
Scope	Full system
Metric Scale	Logarithmic
Node Width	NCO (*Number of created objects by the class*)
Node Height	NCI (*Number of created instances*)
Node Color	NCI (*Number of created instances*)
Edge Width	Number of Creation Between two Classes
Figure	Figure 5

View Intention. The *Creation Interaction* view shows the instance creations between classes of the system during the execution. As described above the width and the color of a class node represents the number of instances created by the class and the height represents the number of instances of the represented class. The *Creation Interaction* view also takes advantage of the embedded spring layout by weighting the springs so that classes heavily instantiating each other classes will aggregate themselves. In addition, the width of the edges represents the number of class-to-class instantiations.

Revealing Symptoms. This view only considers instance level object creations, and may contain:

- Unconnected, tiny, square nodes representing classes that have not been instantiated, and have therefore not been used during the system's execution.
- Connected, tiny, white, square nodes represent classes with few instances. Note that such classes can still instantiate other classes.
- Flat, lightly colored nodes represent classes that heavily create instances, but are not often instantiated themselves. A few objects of these classes create a lot of other objects. Note that we can have an abstract class that still creates a lot of objects simply due to the fact that its methods are used by instance subclasses.
- Narrow, dark nodes represent classes that have been instantiated many times, but whose instances create only few other instances.
- Wide, dark nodes represent classes that have been heavily instantiated and used as the number of methods used and the number of invocations are high.

Case Study. Figure 5 shows the application of the *Creation Interaction* view on an execution of our case study. Four big groups of classes are identifiable:

1. The group on the top of the view, which is composed of a narrow dark node (A) and some flat nodes (B), has an interesting shape. The narrow class node represents the class *AttributeDescription*, a meta-meta-model entity which has been instantiated during the initialization of the system by all the FAMIX meta-model entities. The small flat nodes represent the FAMIX meta-model classes that are not Smalltalk specific but that still have been creating instances of the class *AttributeDescription* to represent themselves during the creation of the FAMIX meta-model.
2. Extracting a source code model is done in two different phases by two different entities: (1) the *VWImporter* (C) which uses the reflective API of Smalltalk to query simple structural information such as classes, methods, attributes, and (2) *VWParseTreeEnumerator* (D) which is a visitor extracting from the abstract syntax tree (AST) more detailed information.
3. The group on the top left of the view represents the first extraction phase where we identify the fact that the big class (C) creates a lot of entities of the surrounding classes (E). The opposite group in the view describes the second phase where the *VWParseTreeEnumerator* (D) creates a lot of instances of the *Access* and *Invocation* classes (F) which are among the most numerous entities in our meta-model.
4. Finally the group on the bottom left reveals an interesting aspect of the system. The big dark node (G) represents the class *Measurement*. Measurements which represents source code metrics are the most numerous entities created during a model analysis. As such they are not represented in memory, but they are stored on file. Instances of *Measurement* are then created during

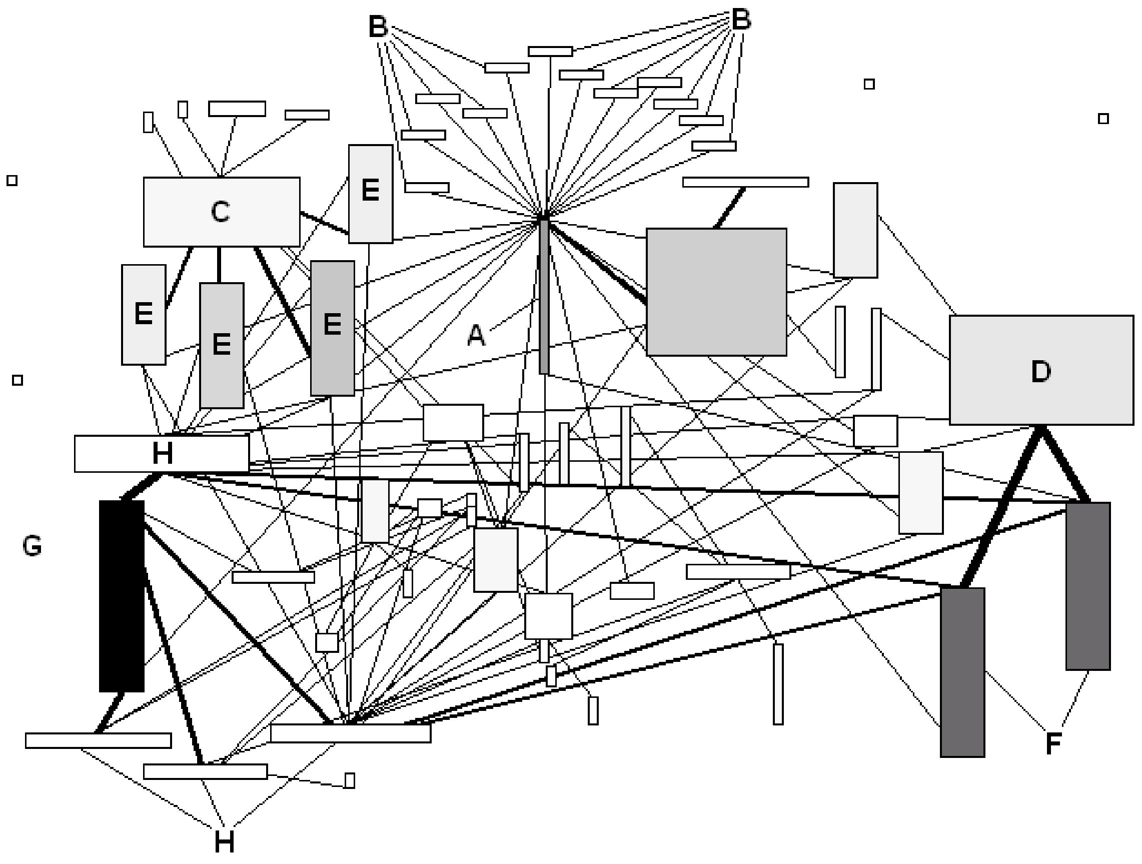

Figure 5. The Creation Interaction view.

the loading of a file like the other entities but a second phase removes them from memory by means of garbage collection. What the picture shows is the fact that during the loading/saving of a source code model, instances of *Measurement* are created. The classes surrounding it are the various classes responsible for the loading and saving (H).

Discussion. The view *Creation Interaction* is clearly more scalable than the *Communication Interaction* view. This is normal as a class has a higher probability to invoke more other classes than to create instances of other classes. The view is useful, as it allows us to classify classes according to their behaviour in terms of the creation of other objects.

5 Discussion

The approach while based on a condensed set of run-time information has proven to be successful to provide insights about the behaviour of an application at run-time. The presented views are rich as they have multiple facets revealing different kinds of information. The approach by its reduction of dynamic information is also applicable to systems that should not be disturbed and for which generating a trace would lead to extremely huge amounts of data. The approach is also incremental in the sense that the collected information can be cumulated. Finally the views provide overviews as well as in some cases more fine-grained information. Our approach could also be plugged dynamically while a system is running. Note that it is linked to the wrapping technology but also to the minimal set of run-time information it requires. Our tool, DIVOOR, uses the Method Wrappers [2] that allows one to dynamically and safely control any method without method recompilation or byte-code modification.

The drawbacks of the approach are that it does not provide fine-grained run-time information at the level of sequence of interactions like sequence diagrams, as done by De Pauw [24] and that it requires the viewer to interact with the view to gather the relevant information. Moreover, our current implementation of the embedded spring layout shows some limits when being applied on densely communicating systems.

6 Related Work

In the past, a great body of research has been conducted to support the understanding of object-oriented applications [12] [17] [21]. Among the various approaches to support the understanding of software behavior that have been proposed in the literature, graphical representations of software execution have long been accepted as comprehension aids. Various tools provide quite different software execution visualizations.

Murphy *et al.* have developed, in AVID, an approach that allows software engineers to specify a high-level model of a system [31]. The software execution can be visualized using these models. Their visualization is oriented towards the liveness of objects and their number. Their work is directed more towards static, architectural models, while our work is more focused on the visualization of different kinds of interactions between classes of a software system during its execution. In a recent version they introduces sampling of events to reduce the information [3].

Systä and Shimba [28] [29] [30] mix static and dynamic information to support program understanding. Shimba uses traces of Java programs and extracts automatically scenario diagrams but also state diagrams representing the runtime of the system based on SCED [15]. Shimba provides also string matching algorithms for recognizing patterns in event traces.

Lange *et al.* with their Program Explorer are focused on views of classes and objects [17] [16]. The authors have developed a system for tracking function invocation, object instantiation, and attribute access. The views show class and instance relationships (usually focused on a particular instance or class), and short method-invocation histories. It is not intended as a global understanding tool. The users must know what they are interested in before they start, whereas our approach is made for covering a complete system.

Jerding *et al.* have created their own interaction diagrams to visualize the entire software execution [10] [11]. The purpose of their tool ISVis is to be able to visualize all the method calls between the classes. They can extract and recognize execution patterns, but its drawback is a certain lack of flexibility in the analysis: It has a good scalability for large numbers of messages, but not for a huge number of classes. In the latter case the visualization becomes less useful.

De Pauw *et al.* presented two different approaches. In their tool Jinsight, they are focused on interaction diagrams [24]. This way, all messages between objects can be visualized. The extraction of execution patterns is also one of the main purposes. However, with a large execution trace it becomes difficult to understand class roles during execution. Earlier on, De Pauw, with its *class call clusters* and *class call matrix* [22] [23], was closer to our approach. These visualizations are simple, they have a good scalability, but they only present a facet of an object-oriented application.

Except the last approach, all of them have in common that they visualize program executions by applying sophisticated diagrams to keep the whole execution trace. In contrast, we extract from the execution trace information which we then condense into a few measurements to enrich our visualizations.

A dynamic program slice [13] is an executable part of a program whose behavior is identical, for the same program input, to that of an original program with respect to a variable(s) of interest at some execution position. Dynamic slicing has been used to support the understanding of program [14]. In a similar way, CodeSurfer [1] supports the understanding by using hypertext facilities but not full visualization support.

7 Conclusions and Future Work

In this paper we presented a new way of presenting runtime information that is not based on a trace of a system, but on a *condensed* and compact set of information, *i.e.,* measurements, extracted during its execution. The measurements are then used to generate run-time polymetric views, simple visualizations enriched with measurements [20].

The views proposed, while being based on simple principles and a condensed set of run-time information, still provide rich insights and highlight multiple facets of the runtime behavior of a system. The advantages of our approach is the fact that it can be applied to systems for which a trace generation would be difficult to extract or too big to efficiently analyze such as webservers or other applications running 24 hours a day.

In the future we plan to extend our current approach in the following ways:

- *Attribute dynamics*. Understanding how attributes are used during the lifetime of an object or a class and their usage frequency is an axis we want to explore.

- *Object lifetime*. Objects do not have the same lifetime over an execution and it would be interesting to identify the different kind of objects.

- *Test coverage*. Understanding and assessing tests is a problem that with the emergence of test-driven methodologies is getting more and more crucial. We plan to apply our approach to understand and estimate tests and their quality. We also think we could verify the coverage of tests and infer which parts of a system are covered by how many tests.

- *Large systems*. We also want to assess whether this approach scales up to deal with traces of very large systems, where the scalability problem is not given by the trace itself, but by the number of involved objects, which has an impact on the visualizations.

Acknowledgments. We acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and Recast: Evolution of Object-Oriented Applications (SNF Project No. 2000-061655.00/1).

References

[1] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Proceedings of WISE'01 (International Workshop on Inspection in Software Engineering)*, 2001.

[2] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98, LNCS* 1445, pp 396–417. Springer-Verlag, 1998.

[3] A. Chan, R. Holmes, G. C. Murphy, and A. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *International Workshop on Program Comprehension*, 2003.

[4] T. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[5] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[6] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In *Proceedings UML '99, LNCS* 1723, Oct. 1999. Springer-Verlag.

[7] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[8] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming*, 12(6):39–44, June 1999.

[9] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[10] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings WCRE*, pp 56 — 65. IEEE, 1997.

[11] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing Message Patterns in Object-Oriented Program Executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, May 1996.

[12] M. F. Kleyn and P. C. Gingrich. Graphtrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88*, volume 23, pp 191–205, Nov. 1988.

[13] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[14] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *5th International Workshop on Program Comprehension (WPC '97)*, pp 80–85, 1997.

[15] K. Koskimies, T. Systä, J. Tuomi, and T. Männistoö. Automated support for modeling oo software. *IEEE Software*, 15(1):87–94, Jan. 1998.

[16] D. Lange and Y. Nakamura. Program explorer: A program visualizer for C++. In *Proceedings of Usenix Conference on Object-Oriented Technologies*, pp 39–54, 1995.

[17] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA '95*, pp 342–357. ACM Press, 1995.

[18] M. Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pp 409–418. IEEE Press, 2003.

[19] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003. Recipient of the Denert-Stiftung Software Engineering Prize 2003.

[20] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[21] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software — Concepts and Tools*, 16:170–182, 1995.

[22] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pp 326–337, Oct. 1993.

[23] W. D. Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proceedings ECOOP '94*, LNCS 821, pp 163–182, July 1994. Springer-Verlag.

[24] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In COOTS '98, pp 219–234, 1998.

[25] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM '2002*, Oct. 2002.

[26] R. Smith and B. Korel. Slicing event traces of large software systems. In *AADebug 2000 International Workshop on Automated Debugging*, 2000. Demo.

[27] I. Sommerville. *Software Engineering*. Addison Wesley, sixth edition, 2000.

[28] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000.

[29] T. Systä. Understanding the behavior of java programs. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pp 214–223, 2000.

[30] T. Systä, K. Koskimies, and H. Müller. Shimba — an environment for reverse engineering java software systems. *Software — Practice and Experience*, 1(1), Jan. 2001.

[31] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, pp 271–283. ACM, Oct. 1998.

[32] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.