Understanding Object-Oriented Programs with Declarative Event Analysis

Tamar Richner, Stéphane Ducasse
Software Composition Group, Universität Bern *

Roel Wuyts
Programming Technology Lab, Vrije Universiteit Brussel†

Abstract

Understanding how components in an application interact to ensure a certain functionality is an essential aspect of understanding a software application. To obtain this kind of information an analysis of the dynamic behavior of an application is more appropriate than a static analysis of the code. Understanding dynamic behavior through event analysis is a challenge because of the large amount of data gathered through program executions. In this paper, we show how we define declarative queries that allow us to filter the event data collected and to define new abstractions which aid in program understanding.

1 Program Understanding through Queries and Views

Program understanding is an important aspect of software engineering. It is essential in the maintainance and reengineering of a system, and especially critical for extending and using frameworks. While documentation should address the program understanding problem, it is often not up-to-date, or missing altogether, and in general does not cover all aspects of an application and as such may not at all address the particular aspect an engineer is interested in.

The best remedy for this lack of appropriate documentation is to allow an engineer to extract documentation from the application for the particular aspect of interest. We believe that a good tool for program understanding should allow an engineer to formulate hypotheses about the structure of the software, and to confirm or reject hypotheses based on evidence from the source code. One such tool is described by Murphy in [MN97]. It allows an engineer to define a high-level model of the software system and a mapping of the source-code onto this model. A 'reflexion model' is then computed which shows how close the high-level model comes to describing the source code. This allows the engineer to iteratively refine the high-level model, so that it better reflects the implementation.

This is a promising approach: an engineer can choose what to query depending on his strategy for understanding the code and on the aspect he is particularly interested in. Each query about an hypothesis gives a view of the software system, and different views can also be combined to get a better understanding of the software [YHC97].

* Institut für Informatik (IAM), Universität Bern, Neubrückstrasse 10, 3012 Berne, Switzerland (richner,ducasse)@iam.unibe.ch, http://www.iam.unibe.ch/∼(richner,ducasse)/
† Pleinlaan 2, 1050 Brussels, Belgium rwuyts@vuab.ac.be, http://progwww.vub.ac.be/∼rwuyts/

2 Understanding Dynamic Behavior

We are interested in how traces from program execution can help in understanding the dynamic behavior of a system. Static source code structures do not lend themselves easily to analysis of behavior: 'what happens when the user clicks on this button?' is information depicted during design through use-cases and scenarios, but later lost in the source code. Tracing events from program executions can help to recover such design artifacts, and to extract information about run-time instances and their interaction protocols which can not be easily derived through a static analysis of the code.

A big challenge in visualizing and analyzing execution traces is the large amount of information generated. Reducing the quantity of information is therefore essential, and can be done at two points: a) at instrumentation: selectively instrumenting the source code to capture only the events related to the aspect to be investigated, and b) at analysis: filtering the events and using abstraction mechanisms on the execution information generated.

In this paper we address the reduction of information at analysis. We present a simple model for dynamic behavior and sketch a tool which allows the user to reduce the amount of information to be viewed and analyzed by defining filters and clustering abstractions. These techniques help in generating different views of the systems behavior.

2.1 A Model for Dynamic Behavior

We propose to model the dynamic behavior of an application using two distinct sets of elements: the set of *run-time entities* which interact with each other to produce a set of *events*. On top of this simple model, we define two operations: *filtering* and *clustering* which can be used to produce new views of the dynamic behavior of the application. The filtering operation allows us to filter out events which are not of interest, and the clustering operations allows us to group entities into a composed entity representing some logical unit, and to group consecutive events into a composed event representing a logical unit of interaction.

```
directMethodEvent(eventNumber,time,senderObject,receiverObject,receivedSelector)
indirectMethodEvent(eventNumber,time,senderObject,sentSelector,receiverObject,receivedSelector)
exitEvent(eventNumber,time,exitOfEventNumber)
```

Simple entities and simple events. As the program executes, three kinds of events (shown above) are recorded to describe an execution trace as a calling stack. The entities visible in the execution traces as receiverObject and senderObject are considered *simple entities*. As we are working in Smalltalk, where classes are also objects, a simple entity is either a class or object. A *simple event* is either a message send of one *simple entity* to another, directMethodEvent or indirectMethodEvent, or an exit event representing a return from a message send, exitEvent. We then define two operations:

Filtering. Applied on events, this operation allows us to filter out events from the execution trace using a variety of criteria. Some examples of filtering are:

- removing events representing self sends
- removing events representing sends to or from a specific entity
- removing events of a certain kind (e.g. all initialize, create events, accessors)

Clustering of events. Events can be clustered by grouping a set of consecutive events into a *composed event* which represents a logical unit of interaction. Some examples of possible composed events are:

- a sequence representing a repetition of an event (e.g. loop)
- a sequence of events below a certain calling stack level can be grouped to form one event e.g. if (m1 (m2 (m3)) (m4)) is a calling stack, m23 = (m2 (m3)) could be grouped together so that the new trace is (m1 (m23) (m4))
- a sequence of specific messages exchanges representing a recurring scenario

A clustering of events groups a number of events into a logical unit and hides the events it is composed of.

Clustering of entities. This operation can be applied on simple entities to create *composed entities*. Entities can be grouped together according to different criteria:

- clustering of classes:
  - subclasses
  - enumeration of the classes to group
  - aggregation : a class and its aggregate classes
  - grouped by some other criteria
- clustering of objects:
  - all instances of the same class
  - same as for classes but at the level of instances

Note that a clustering of entities automatically defines a filter which removes the events occurring between those entities.

2.2 A First Experiment

To generate a dynamic trace of Smalltalk applications, we instrument the code using Method Wrappers as used in the interaction diagram tool [BJRF98]. The execution of the application then generates a collection of events, representing the dynamic behavior involving the instrumented methods using three kinds of events: *direct event*, *indirect event* and *exit event*. Direct events correspond to the reception by an object of a message from a sender. Exit events represent a return from an invoked method. Indirect events are conceptually similar to direct events, but represent messages sent by an instrumented object to another instrumented object but via a non-instrumented one.

We use a Prolog-like language integrated in Smalltalk, SOUL [Wuy98], to represent these events as facts and to define queries on them. The following text shows a short sequence of events corresponding to the creation of a counter and its initialization. The simple entities which are visible here are the classes CounterView and CounterMVC, and the object @0 (instance of CounterMVC).

```
FACT directEvent([1],[266339],[nil],[CounterView],[#open])
  FACT directEvent([2],[266340],[CounterView],[CounterView],[#defaultCounterClass])
  FACT exitEvent([3],[266341],[2])
FACT directEvent([4],[266342],[CounterView],[CounterMVC],[#new])
  FACT directEvent([5],[266342],[CounterMVC],[@0],[#setToZero])
    FACT directEvent([6],[266342],[@0],[@0],[#count:])
      FACT directEvent([7],[266343],[@0],[@0],[#count:])
      FACT exitEvent([8],[266343],[7])
    FACT exitEvent([9],[266344],[6])
  FACT exitEvent([10],[266344],[5])
FACT exitEvent([11],[266344],[4])
FACT directEvent([12],[266344],[CounterView],[CounterView],[#openOn:])
```

We then define rules which can be used to query the repository of facts. Simple rules allow us, for example, to query about all the classes involved in the trace behavior, retrieve instances of a certain class and check all senders of a specific message. Since SOUL has been used to construct a declarative framework that allows reasoning about the structural aspects of code, it offers an advantage over Prolog in that it allows us to couple dynamic and static information in queries and rules without having to reify static information as facts.

The following rules shows how a rule for obtaining the public interface of a class is formulated by combining the notSelfSend and the invokedSelectorOnClass rules.

```
publicInterface
''returns true if there is an instance of ?class which receives
?receivedSelector from another object
Rule
  head: publicInterface(?class,?receivedSelector)
  body: notSelfSend(?number,?sender,?receiver,?receivedSelector),
        ''notSelfSend returns true if in event ?number the ?sender is not the same
          object as the ?receiver''
        invokedSelectorOnClass(?receivedSelector,?receiver,?class)

invokedSelectorOnClass
''returns true if ?selector is invoked on ?receiver and ?receiver is an
instance of ?class''
Rule
  head: invokedSelectorOnClass(?selector,?receiver,?class)
  body: invokedSelector(?selector,?receiver), class(?class), [(?receiver class) = ?class ]

invokedSelector
''returns true if ?selector is invoked on ?onReceiver''
Rule
  head: invokedSelector(?selector,?onReceiver)
  body: directEvent(?number,?time,?sender,?onReceiver,?selector).
Rule
  head: invokedSelector(?selector,?onReceiver)
  body: indirectEvent(?number,?time,?sender,?sentSelector,?onReceiver,?selector)
```

The next example shows the createEvent rule which defines an event corresponding to a request to a class to create a new instance. It makes use of two new rules. The creationEvent rule matches an event in which a method is invoked on a class, where the method does a 'self new'. The creationEventIn rule matches a method invocation on a class where within this method invocation an event occurs which is a creationEvent.

```
createEvent
''case where a creationMethod i.e. 'new' is invoked on a class''
Rule
  head: createEvent(?number,?sender,?receiver,?receivedSelector)
  body: notSelfSend(?number,?sender,?receiver, ?receivedSelector),
    creationMethod(?receivedSelector).

''case where a method is invoked on a class, where a 'self new' is done within
the method''
Rule
  head: createEvent(?number,?sender,?receiver,?receivedSelector)
  body: notSelfSend(?number,?sender,?receiver, ?receivedSelector),
      creationEvent(?number,?sender,?receiver,?receivedSelector).

''case where a method is invoked on a class, where lower in the calling stack
a method is invoked, where a 'self new' is done within the method''
Rule
  head: createEvent(?number,?sender,?receiver,?receivedSelector)
  body: notSelfSend(?number,?sender,?receiver,?receivedSelector),
      class(?receiver), exitOfSend(?number,?exitNumber),
      creationEventIn(?number,?exitNumber,?receiver,?receiver,?selector)
```

The creationEvent rule shows how static information reified in the SOUL framework can

be used in combination with dynamic information. Furthermore, SOUL allows us to use blocks with Smalltalk predicates, simplifying the definition of rules.

```
creationEvent
''returns true is a method is invoked on a class, where a 'self new' is done
in the code of that method''
Rule
  head: creationEvent(?number,?sender,?receiver,?receivedSelector)
  body: directEvent(?number,?time,?sender, ?receiver, ?receivedSelector),
      class(?receiver), metaClass(?receiverClass), [ (?receiver class) = ?receiverClass ],
      methodInClass(?receiverClass, ?compiledMethod, ?receivedSelector),
      statements(?compiledMethod, ?statList),
      isSendTo(variable(¯"self"¯), ?creationMethod, ?statList),
      creationMethod(?creationMethod )
```

2.3 Filtering and Clustering

Queries such as the ones shown in the previous section can be used to define filters on the sequence of events. A filter is basically a query which returns a list of events - either a list of events we are interested in, or a list of ones we are not interested in. The result of such a query is then used to update the repository of events to reflect only the events of interest.

We have not yet implemented clustering operations. A clustering of events is based on a query which returns a list of consecutive events to be grouped together. If a send event is represented by S(and an exit by) consider the execution trace: S1(S2(S3() S4() S5())). If S2(S3() S4() and S5()) are to be clustered, the resulting trace would be S1(S2composed()); if S4() and S5() are to be clustered the resulting trace would be S1(S2(S3() S45composed())). Composed events must be given an identity and a description which relates them to the original execution trace. To implement the clustering of entities a grouping of run-time entities must be defined, together with a filter which removes all message exchanges between members of the group. We expect here to use queries which return a list of run-time entities to define different kinds of clusters. Composed entities must also have an identity so that they can be reasoned about just as simple entities.

Although our focus is not on visualization techniques for dynamic information, we are using an interaction diagram tool [BJRF98] to display the events (also known as scenario diagrams, temporal message flow diagram), and the operations we define in our model clearly have a visual interpretation in such diagrams. The filtering operation removes some of the events to be displayed. The clustering of entities groups together run-time objects in a single time-line, effectively removing all events between the run-time objects belonging to the group. The clustering of events would group a sequence of events into one logical unit, visually displayed as one event. We intend to enhance the interaction diagram tool with these operations.

3 Discussion and Future Work

Several researchers have looked at visualizing and analyzing dynamic information from execution traces as an aid to program understanding. [PKV94] describe a way of modeling the execution of object-oriented programs which allows for a variety of visualizations displaying statistics (e.g. interclass call matrix by frequency, histogram of instances, allocation matrix) derived from program executions. Sefika et al. [SSC96] combine static and dynamic information in comparing graphs which display relationships extracted statically (e.g. weighted inter-class call graph) to graphs summarizing information generated from dynamic executions (e.g. affinity diagram showing dynamic inter-class call relations). Their views can display the interactions of architectural units such as subsystems, but this requires an 'architecture-aware instrumentation' and necessitates an a priori understanding of the program structure.

The tools mentioned above focus on visualizing summaries collected from dynamic executions. The Scene tool [KM96] on the other hand, retains the execution events to display interaction diagrams and offers hyperlinks connecting the scenario diagrams to the source code and other development artifacts. In Program Explorer [LN95] static information and information about dynamic events is reified as prolog facts, allowing for the coupling of static and dynamic information in checking for the presence of design patterns in code. ISViS [JR97] is a visualization tool which displays program execution using interaction diagrams and also offers several tactics to aid in program understanding. Interactions can be removed or grouped into scenarios and actors (function, object or data item) can also be grouped. The tool also offers pattern matching capabilities to aid in identifying repeated patterns of events.

Our work on dynamic analysis is closest to the work of Lange [LN95] and Jerding [JR97] and is inspired by the work of Murphy [MN97] in using static information to compute 'reflexion models'. Our goal is to allow a general querying of dynamic information coupled with static information. This querying facility should allow an engineer to define a range of filters and clusters and to combine these to generate different views of the dynamic execution. We would like to allow an engineer to describe a view of the software behavior using composed entities and composed events and to be able to compute a 'reflexion model' for this architectural view.

We are currently working on several aspects of a tool for dynamic analysis:

- defining and implementing clustering operations,
- achieving more flexibility at instrumentation,
- experimenting with larger applications to discover useful filters, and
- enhancing the interaction diagram tool to incorporate filtering and clustering

4 Acknowledgements

This work has been funded by the Swiss Government under NFS grant MHV 21-41671.94 (to T.R.), Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT programme Project no. 21975.

References

[BJRF98] John Brant, Ralph E. Johnson, Donald Roberts, and Brian Foote. Wrappers to the rescue. In *To appear in Proceedings of ECOOP'98*, LNCS, page ??? Springer-Verlag, 1998.

[JR97] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *Proceeding of WCRE '97*, pages 56–64. IEEE, October 1997.

[KM96] Kai Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active test for illustrating object-oriented programs. In *Proceedings of ICSE-18*, pages 366–375. IEEE, March 1996.

[LN95] D.B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357, 1995. mixin dynamic and static information for model capture.

[MN97] Gail Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 17(2):29–36, aug 1997.

[PKV94] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[SSC96] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring complicance of a software system with its high-level design models. In *Proceedings ICSE-18*, pages 387–396, March 1996.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *To appear in Proceedings of TOOLS USA'98*, page ???, 1998.

[YHC97] A.S. Yeh, D.R. Harris, and M.P. Chase. Manipulating recovered software architecture views. In *Proceedings of ICSE'97*, 97.