A Design Virtual Machine for Static Analysis of Smalltalk

TR 29.3021

Stephen L. Burbeck

Stephen G. Graham

Emerging Technologies
IBM Network Computing Software
Research Triangle Park, North Carolina 27709

The authors have used their best efforts in the preparation of this report. International Business Machines Corporation and the authors make no representation or warranties with respect to the accuracy or completeness of the contents of this report.

The following terms, denoted by ® or ™, used in this report are registered trademarks or trademarks of International Business Machines Corporation in the United States and other countries.

IBM
VisualAge

The following terms, denoted by ® or ™, used in this report are registered trademarks or trademarks of the listed holder. All other brand names and product names used in this report are trademarks, registered trademarks, or trade names of their respective holders.

Java	Sun Microsystems, Inc.
Pentium	Intel Corp.

© Copyright 1998 by International Business Machines Corporation. All rights reserved.

ABSTRACT

This report describes a method for synchronizing fine-grain design descriptions of method level collaboration in Smalltalk with the code that implements those collaborations. The design intent is described for each method by a method *signature* that specifies the kinds of objects intended to be passed as method arguments and to be returned by the method. The validity of these descriptions in the context of the method's code is determined by a static analysis of the code. This process is related to but far more challenging than type checking in a strongly typed language. We implement static analysis by compiling the code to a bytecode-like description that is "executed" by a design-level virtual machine. The static analysis of a typical method takes less than 100 milliseconds on a modern PC and therefore can be done in real-time whenever any method is browsed or changed. Any mismatches between code and design are signaled immediately to the user. This allows a Smalltalk developer to iteratively and constantly ensure that the design descriptions match the code.

It is assumed that the reader is familiar with programming in Smalltalk.

ITIRC KEYWORDS

- Executable design
- Virtual machine
- Object oriented design
- Signature
- Data types
- Smalltalk

CONTENTS

INTRODUCTION

Synchronizing code with design

The map is not the territory. The blueprint is not the building. The design is not the program. We build abstract analogs such as maps, blueprints, and software designs because they are useful for explaining, navigating, and understanding the richer underlying realities they represent – at least they are useful when they capture the right abstraction of the underlying realities and when they accurately correspond to those realities.

With software it is rare for even the most general design of an implemented system to be either complete or accurate. In many projects the senior programmers brainstorm on a whiteboard, produce the program, then produce just enough of a retrospective design to satisfy their management. In projects with formal analysis and design stages, the design may be accurate when it is first made, but it seldom if ever matches the final implementation. As code is developed it diverges from the design either by accident or by necessity. These changes are rarely transferred back to the design documents because programmers seldom take the trouble to find the design documents and to edit them.

The lack of accurate design adds dramatically to the lifecycle cost of software systems. Mismatches between design and code slow initial development of large systems because teams working on one portion of the system rely in part upon the design descriptions of other portions of the system. Inaccurate design has an even more dramatic effect on maintenance because maintenance done without understanding the underlying design is time consuming, prone to error, and encourages the accumulation of unwanted complexity in the system [Bur96]. These problems go a long way toward explaining the common assertion that seventy percent of the lifecycle cost of software is in maintenance.

Tools that help to keep design and code synchronized have the potential to substantially reduce lifecycle costs. That observation motivated many CASE tool vendors to attempt to provide tools that allow one to create and maintain elaborate designs from which the production code is generated. Their efforts were misguided because they did not recognize the fundamental differences between code and design.

This report describes an approach for keeping code and design synchronized in Smalltalk systems.

Elements of OO Design

Object-oriented design involves just a handful of basic elements. The most familiar are:

- Object Structure – diagrammatic specification of which objects are referenced by or collaborate with which other objects.
- Class Structure – diagrammatic or textual descriptions of inheritance and attribute relationships between classes.
- CRC (*C*lasses, *R*esponsibilities, and *C*ollaborators) – short textual descriptions of the responsibilities of each class and a list of the other classes that are its collaborators.
- Object Interaction (collaboration) – diagrammatic and textual specification of the responsibilities that are invoked and the messages that are sent to other objects to collaboratively accomplish a given function.

Other design elements, such as state-transition diagrams, are used far less often and will not be discussed here.

Relationship between design elements and code

Design and code are a bit like Siamese twins. They are neither completely separate nor completely one. Design and code cannot be treated as if they were completely separate despite the desires of those who wish design to be “language neutral” because language features, e.g., multiple inheritance, can have dramatic effects on design. Nor can they be treated as one despite the desires of the CASE vendors. They overlap in that they both describe the same system but they are different because the intended audiences for those descriptions are quite different.

Design communicates the intent of the designers (and of the program) to other humans. Communication of design relies on the human ability to interpret graphic layouts and the semantics of textual descriptions. Designers rely on the fact that humans share a vast common knowledge and can deal with abstraction and generality even though we are poor at handling masses of detail. Code communicates design intent to the machine. The machine is not hampered by masses of detail but is oblivious to graphical layout, textual semantics, abstraction and generality.

The major elements of OO design differ in the degree to which they document abstractions and generalities and in the degree to which they rely on the semantics of natural language. So they are reflected quite differently in code and therefore present quite different challenges to those who would keep design and code synchronized.

Classes and their inheritance and attribute structure

The structure of the inheritance relationships between classes is reflected directly in Smalltalk class descriptions and in the structure of instantiated classes in the image. It is a simple task for tools to translate this information into accurate inheritance diagrams. However, most Smalltalk systems provide no formal way to explicitly specify what kinds of objects are intended to occupy those attribute variables (i.e., unlike Java™ and C++, Smalltalk does not have type declarations). All that is required to generate attribute structure diagrams is the addition of some sort of “type” information for attributes. The *qualifiers* described subsequently in this paper can play this role.

Responsibilities

Responsibilities are short, pithy, active-verb phrases that describe an object’s behavior in natural language. Not surprisingly, traditional Smalltalk systems do not support the notion of responsibilities. At best, responsibilities might be included in class comments. One experimental Smalltalk development environment (demonstrated at OOPSLA’93 by Knowledge Systems Corp.) included tools to better integrate responsibilities with the code and to help maintain them as the system is changed. However, these tools never became commercially available.

Object interaction or collaboration

Collaboration – the client-server relationship between objects that are implemented by messages sent from one object to another – is the element of OO design that is most important for understanding, maintaining, harvesting and reusing an OO system. Collaborations are also the most difficult to deduce by reading code. Each message specified within a method ***is*** a method level collaboration. If the fine grain design were complete, i.e., if the entire system were described by object-interaction diagrams, each message described in the code would be represented by a method level collaboration described in the design. However designs are never completed to such an exhaustive level of detail and even if they were, they would be more unwieldy than useful.

Collaborations are difficult to deduce from code because code describes the interactions (i.e., the messages) that will take place between the objects involved in its execution but does not explicitly specify the identity of these objects or their placement in the inheritance hierarchy. To use a theatrical metaphore, each method is the script for a portion of a scene. The code specifies the lines the actors speak. What is missing in the code is a record of the designer’s intent in casting the actors in method-level collaborations. That is, just *who* are these actors and what are their roles in the larger play? This missing information represents a substantial portion of the knowledge needed to understand, reuse, or to modify an object-oriented system. We describe in this report how these collaborations can be better documented, i.e., how we can specify the actors, and how the design descriptions of the collaborations can be kept synchronized with the code.

COLLABORATION IN SMALLTALK

Critics of Smalltalk's lack of "strong" typing seem to imagine that collaboration in Smalltalk is akin to anarchy. Despite the theoretical freedom allowed by Smalltalk, both the design and the implementation of any realistic Smalltalk application is not nearly so unconstrained. With rare exceptions, the objects intended as collaborators in a method are quite specific, as is the object to be returned by the method. The problem is that the language has no support for declaring these intentions explicitly and the information from which one could deduce this design intent for the collaborators may be distributed unpredictably throughout the system. By convention, some of these intentions are expressed by variable names, e.g., a method argument named "anInteger" is intended to be some subclass of Integer. But this convention isn't always sufficient to describe design intent, isn't always followed in any case, and does not provide a way to declare what is returned from a method. We therefore have proposed a syntax for specifying this kind of information for each method in method *signatures* [BG98] (we describe signatures briefly in a later section). Signatures specify the design intent for collaborators such as method arguments and return values.

The variety of collaborators

In Smalltalk, unlike in hybrid languages such as C++, *everything* is an object. Numbers, characters, arrays, and strings are objects with behavior that can be extended by the programmer. Classes themselves are objects that exist and provide important function at runtime. Even control structures are implemented with objects – blocks – that may be executed at will. Classes and blocks, like any other object, may be assigned to variables, passed as arguments to methods, or returned from methods. So, the notion of a collaborator in Smalltalk is far more powerful than the notion of a collaborator in C++. Standard type systems, such as the ones in C++ or Java, cannot adequately describe the variety of collaborators that commonly are used in Smalltalk. The signature system we use to describe collaborators deal with all these issues and more.

It should be noted that the problem of describing the variety of collaborators in Smalltalk systems bedevils designers as well as programmers. Popular object oriented design methodologies provide no good way to deal with collaborations richer than those found in C++. In part for technical reasons and in part for marketing reasons, they aim to be "language neutral" (in reality this means they support only the least common denominator). Thus designs that rely on class behavior at runtime or blocks being passed from one method to another do not fit well with these methodologies. Perhaps the growing popularity of Java, which has constructs such as interfaces, inner classes, and some support for introspection, will cause OO methodologists to expand their notions of collaboration.

Sources of collaborators

The collaborators used by a given method enter the execution scenario of a method in three ways:

- They may be objects held by the receiver (in instance variables), its class (in class variables and class-instance variables), pools of classes (in pool variables), or globally (in the Smalltalk dictionary). These collaborators are the ones specified by *attributes*.
- They may be objects passed into a method as arguments. The *senders* of the message that invokes the method determine these collaborators.
- They may be objects returned by messages sent to other collaborators. The *implementors* of the messages sent by the method determine these collaborators.

Thus the information needed to deduce all of the collaborators of a method may be widely distributed. Changes to any of this distributed code may change the collaborators of a method. Does this mean that documentation of collaboration is an impossible task? No. But the distributed design information, if it is to be maintained as code is changed, must be co-located with the code that it describes and the design information must have a granularity commensurate with individual methods. Tools, then, are needed to help spot mismatches between the code in a given method and the distributed design that may impact that code.

ALTERNATIVE APPROACHES FOR SYNCHRONIZING CODE AND DESIGN

Automated production of code from design

One approach supposes that, if programmers are unable or unwilling to keep the code synchronized with the design, perhaps we can dispense with programmers and simply generate the code from the design. Then the system can be enhanced and maintained at the level of the design. This has been the holy grail of CASE tool vendors.

Strengths

In some simple cases, especially when the application merely maintains a database, this approach works. When it does work, maintaining design is generally quicker and more reliable than maintaining code.

Weaknesses

For general OO programming this approach fails for at least the following reasons:

- Lack of omniscience – Analysts and designers seldom, if ever, anticipate all the details encountered in the actual coding. Programmers must make changes that extend or “violate” the design because they discover relationships or cases not foreseen by the designers. Removing the programmers from the process does not impart omniscience to the designers.
- Need for algorithms – Most real-world applications contain behavior that is best described with algorithmic expressions. Programming code constructs have evolved to effectively and efficiently express such algorithms. Calling a detailed description of program behavior “design” simply because it is expressed in a formalism that isn’t recognizable as “code” does not eliminate the complexity of the behavior. In fact, expressing complex algorithms in some “design” language will likely make them seem even more complex.
- As discussed in the previous section, OO design methodologies have not progressed beyond the limitations of C++. So common Smalltalk practices cannot even be expressed in these methodologies

The fact is that software design is not intended to address all the detailed issues that are confronted in coding any more than architectural blueprints are intended to address the order in which every structural member is added to the frame of a building, or the exact placement of every nail.

Automated extraction of object structure from code

Some tools are available that can create more or less detailed object structure diagrams directly from C++ class definitions that contain inheritance and attribute type information. Smalltalk systems usually do not document attribute "type" information. Without the attribute information, tools can only extract the inheritance structure.

Strengths

The only real value of this method is its simplicity and reliability. Automated extraction of object structure information replaces manual maintenance of object structure diagrams (e.g., with a drawing tool) that is tedious and prone to error.

Weaknesses

Despite the tendency to discuss this approach as if it were extracting design from code, it does not actually parse and model code other than C++ header files or Smalltalk class definitions. Therefore this approach can at best identify "has-a" and "is-a" relationships. These relationships may imply collaborations of a structural sort, but they do not identify any of the transient collaborations that are important for understanding design. Nor does this approach provide any information about algorithms.

Automated deduction of design by analyzing code execution

Since collaborations implicit in Smalltalk code are difficult to deduce statically from the code and may not be fully determined until runtime, perhaps we should collect the collaboration information at runtime. After all, Smalltalk is strongly typed at runtime so we can determine exactly what kinds of objects are participating in all collaborations by examining the receiver and the arguments involved in all message sends. This is the premise of tools that "spy on" the actual objects and messages sent at runtime (e.g., the Ovation tool built in IBM® Research, and the prototype shown at OOPSLA '94 by The Object People). The resulting information can be used to specify the collaborations observed during the execution.

Strengths

This approach is unique in that it captures specifics of the collaboration design without parsing and analyzing code. And it can't be "fooled". The objects seen to be collaborating at runtime really do collaborate. In principle, a great deal of information can be collected for each message send:

- the class and instance ID of the receivers and the args
- the method or block context in which the message send occurs
- the class that implements the method dispatched as a result of the message send

This sort of information can be used to analyze how specific instances collaborate with other specific instances. And the order in which messages are sent provides information (albeit difficult to interpret) about algorithms.

Weaknesses

At first glance, this approach would seem to entirely solve the problem of synchronizing code and design. However it suffers from the following problems:

- It requires test cases to exercise the code. Each of these test cases must construct an initial state that may be quite elaborate. Small differences in the construction of a test case may result in a large difference in the collaborations that appear in the execution. So the test cases require careful construction and can easily become obsolete as the system changes. The effort needed to construct and maintain these test cases is a deterrent to routine use of this technique.
- Full coverage by the test cases is difficult to obtain and the degree of coverage is difficult to assess. This undermines confidence in the resulting design. Without full coverage, the extracted collaboration design is likely to be incomplete in important ways. For instance, the way a system is designed to handle the exceptional cases can be more telling than the way it handles the common ones.
- It can be difficult to separate the wheat from the chaff. The interesting messages may be sprinkled sparsely among a large volume of uninteresting messages. Visualization tools such as Ovation may help this problem.
- Runtime collaborations are often too specific. In many cases the design is properly expressed in terms of collaborations between abstract superclasses. This is especially true with frameworks. But abstract classes may not show up at runtime.
- The effort and time needed to revise the design when changes are made to the code tend to be proportional to the size of the application not to the size of the change. That is, the test cases exercise relatively large chunks of the system. Relatively small changes to the system may have a wide enough impact on the design to require many test cases to be reanalyzed. So synchronization of code and design will generally be a batch process. Programmers will not in general have access to up-to-date accurate design.

Local static analysis of code based upon fine grain design declarations

Many of the weaknesses of the above approaches derive from a unidirectional focus. They attempt either to infer code from design or to infer design from code. Yet design and code address different issues. Code inferred from design is usually poor code and often incomplete. Design inferred from code is similarly flawed. Rather than dispense with humanly created code or design, we recommend that both should be maintained in such a way that the declared design intent remains synchronized with the actual effect of the code.

What sort of design might we reasonably expect to keep synchronized with code? It is not practical to specify the global collaboration web of any substantial Smalltalk application. That web is far too complex and is affected by far too many forces of change. Yet it *is* practical to ask for piecemeal design descriptions, i.e., design localized to single methods. Ultimately, collaborations are defined by individual methods, each of which has a relatively small number of collaborators. It is reasonable and practical to ask the designer/programmer to characterize the attributes of each class and, for each method, to characterize the arguments and the returned object.

This method-by-method design information is sufficient for a static analysis to determine which methods might be invoked by messages sent to collaborators in the code. Since each of these methods contains similar design information, the analysis tools can determine the objects returned from message sends, which may later become receivers of other messages. These deductions about collaborations can then be compared to the fine grain design declarations. Mismatches are signaled to the user. Because *both design and code are artifacts of human intent*, it is the user's responsibility to decide whether the design declaration should be changed to match the code or vice versa. That is, static analysis serves to ensure the validity of the design but it does not substitute mechanically inferred design for human intent. Since the static analysis is local to the method, it is quick (usually less than 100 milliseconds for typical methods on a Pentium™ ThinkPad™). Thus validation can be done each time a method is changed or even viewed in a browser. The cumulative effect of providing the developer with method-by-method static analysis feedback is that the design and the code is iteratively and continuously kept synchronized.

This little-known sort of static analysis was first prototyped by Borning & Ingalls [BI82]. Another system was developed by Knowledge Systems Corporation (unpublished) in a Smalltalk development environment that was shown at OOPSLA'92 and '93. Neither system was made generally available. But they demonstrated that local static analysis can provide a practical way to keep design and code synchronized.

The present work is based on a similar local static analysis approach. Its novelty is in the way that the static analysis is done.

Strengths

This approach explicitly recognizes that both design and code are important artifacts in their own right. Design can be expressed as it properly should be, e.g., in terms of abstract classes where appropriate. Detailed collaboration information is expressed in its most natural form (code) yet can be viewed from a design perspective, e.g., as Object Interaction Diagrams or Object Structure Diagrams when desired [GB98d, GB98e]. Other strengths are:

- Because mismatches between design and code are detected each time a method is viewed or changed, the issues are fresh in the mind of the developer. Correcting mismatches is usually quick and easy.

- The local static step-by-step analysis of the code in the context of fine-grain design provides a means of validating the internal consistency of the design and detecting errors in the code that otherwise would result in “doesNotUnderstand” run-time errors.
- It does not require test cases to exercise the code.
- The effort and time needed to revise the fine grain design when changes are made to the code is approximately proportional to the size of the change. True, a change can be made to a single method that invalidates the design of many other methods that use or are used by the changed method. But if it invalidates the design of those methods, it probably also invalidates their code. So those methods must be examined and corrected if necessary whether or not design is being maintained.

Weaknesses

For this approach to be effective, the fine-grain design information must be made available for all classes and methods that an application may use. Thus it requires that the fine-grain design be “reverse engineered” for all prior work on the application as well as for much of the standard Smalltalk image – a worthwhile but laborious task. We have done this for most of the commonly used classes in IBM VisualAge®.

Since this is a largely unexplored approach, the limits of static analysis are not yet fully understood. There is a tradeoff between the expressiveness of the “type” system (see next section) and the power of the static analysis technique. Weaknesses in analysis can be compensated for by additional detail in the fine-grain design declarations. Some possible problems are:

- Certain code constructs (e.g., #perform:) can frustrate the ability to deduce the type of the returned object.
- Non-standard flow-of-control such as that invoked during exception handling may not be traceable.
- Difficult issues arise when some, but not all, of the possible collaborators understand the message sent.
- Local analysis of abstract methods (i.e., methods implemented in an abstract class that are intended to work only when invoked in a concrete subclass) can be difficult unless information is available about which concrete subclasses are intended to receive the message.

DOCUMENTING FINE-GRAIN DESIGN

Method Signatures, Qualifiers, and Attributes

In the above section, the term “fine-grain design” was used without definition. It refers to design information that specifies the kind of objects involved in a method’s execution. Each method is described by a *signature*. A signature defines in a formal way the kind of object intended to be returned from the method and, if the method has arguments, the kinds of objects intended to occupy the arguments [BG98a].

The “kind of object” is specified by a *qualifier*. A signature is an organized collection of qualifiers delimited by angle brackets. By convention, it is entered into each method as a formatted comment that is inserted between the method’s message pattern and the normal method comment. Signatures may also be kept elsewhere, especially in cases where the development environment restricts editing of preexisting code (e.g., Kernel classes). Variables other than method args, e.g., instVars or globals, also need qualifiers. Each of these is modeled by an *Attribute*.

Signature Format

For unary methods (which do not have arguments) only the return qualifier is specified:

< ^ *qualifier*>

For methods with one or more arguments, the signature adds a qualifier for each argument keyed by the argument’s name:

< *arg1Name: qualifier, arg2Name: qualifier, ...,* ^ *qualifier*>

The following example method, Magnitude>>between:and:, illustrates some common kinds of qualifiers: those that place an object in an inheritance hierarchy, those that specify certain special objects (e.g., true) and those that include a list of alternatives.

```
between: min and: max
      "<min: hierarchyOf Magnitude, max: hierarchyOf Magnitude,
       ^(true | false)> "

      ^(min <= self) and: [self <= max]
```

The signature for this method asserts that min and max are expected to be kinds of magnitudes and the method returns one of two alternatives: true or false.

Qualifiers

A *Qualifier* characterizes the objects that are qualified to occupy a variable given the role the variable plays in the design of the method. As such, the system of qualifiers proposed here is an OO type system from the clients' viewpoint [Den91]. We use the term *qualifier* rather than *type* to avoid some of the confusion and debate about just what is an OO type.

The most commonly encountered OO type system are the ones in Java or C++. C++ types are relatively simple because messages may be sent only to objects that inherit from a common class, collections may hold only objects that inherit from a common class, and classes don't exist at runtime. Java is somewhat more polymorphic in that it adds the notion of interfaces that allow messages to be sent to an object if the message is specified in its interface. Smalltalk is fully object-oriented and fully polymorphic. In Smalltalk any object that understands the necessary messages may take part in a collaboration without regard for inheritance. Smalltalk collections can accommodate elements of arbitrary (and varying) types, and Smalltalk collaborators may be classes themselves. All of these issues are handled by the following qualifier system. (Note, signatures are described in more detail in [BG98], however the following list includes a few qualifiers needed for static analysis that are not described in that document.).

Class Qualifiers

- ***h**ierarchy**Of** AClass* – instances of AClass or one of its subclasses (abbreviated as *hOf* AClass)
- ***i**nstance**Of** AClass* – instances of AClass only (abbreviated as *iOf* AClass)
- ***h**ierarchy**Of** class AClass* – AClass itself or one of its subclasses (abbreviated as *hOf* class AClass)
- *class AClass* – AClass itself

Special Qualifiers

- *any* – any object is valid (equivalent to hOf Object in a single rooted system)
- *none* – no object is valid (used as a return qualifier in a method that does not or might not return, e.g., because of an exception)
- *nil, true, false* – these qualifiers refer to the exact objects specified

Instance Qualifiers

Almost all qualifiers specify no more than a general property of the object (usually its class). However Smalltalk code involves many messages to self that return self, or in some cases a method returns one of its args (e.g., Collection>>add:). In such cases it is important to be able to specify in the design that the returned object is self or an arg. A corollary is that it also becomes necessary to distinguish between self and a copy of self. These issues are handled by the following qualifiers.

- *self, myClass* – The *self* qualifier indicates the receiver of the message; *myClass* indicates the class of the receiver. Examples:

Code	Appropriate Qualifier
`^self`	*^self*
`^self new (in a class method)`	*^hierarchyOf self*
`^self class`	*^myClass*
`^self class new`	*^hierarchyOf myClass*

- *arg1, arg2, ...* – these qualifiers refer to the specific instances passed to the arguments of the method in which the signature appears.

s*elf copy* – refers to an object of the same class as self, but not the same instance.

argN copy – refers to an object of the same class as the Nth arg, but not the same instance.

Numeric Return Qualifiers

# – a return qualifier to be used in arithmetic methods to indicate that the class of the return is determined from numeric generality rules applied to the receiver and the method arg. This qualifier is used primarily for signatures in the Number hierarchy.

Qualifier Aspects

qualifier {aspectName1: qualifier, aspectName2: qualifier, ... }

Aspects are used with aggregate objects such as collections to specify the kind of object held by the aggregate. If a method accepts a collection as an argument and makes any use of the objects in the collection, the qualifier should use an aspect to specify the kind of object assumed to occupy the collection. Similarly, if a method returns a collection, the return qualifier should specify via an aspect what is known about its elements.

The predefined aspects and their defaults are:

Qualified Object	Predefined Aspects	Default Aspect Qualifier
Collection	*of:*	*any*
Dictionary, Association	*key:, value:*	*any*
Point	*x:, y:*	*hierarchyOf Number*
Stream	*on:*	*hierarchyOf String*

We state elsewhere [BG98] that the user may define new aspects. However aspects other than the above are not supported by the static analysis system

Alternative Qualifier

(qualifier **|** *qualifier* **|** *...)* – at least one of the qualifiers must apply

Alternative qualifiers are used in cases where the behavior expected of an object is not provided by a common superclass. One surprising example is the use of *(true* | *false)* instead of *hOf Boolean*. In VisualAge, Boolean does not implement all of the messages implemented by its two subclasses: True and False. Thus static analysis may find spurious errors if an object is declared to be *hOf Boolean* that would not be found if *(true* | *false)* is used.

Block Qualifier

[***:b****lockArg****1*** *qualifier,* ***:b****lockArg****2*** *qualifier, ..., ^ qualifier]*

Since the return result of a block may be one of its block arguments, the return qualifier inside a block may refer to its arguments, e.g., ^ *b1* or ^ *blockArg1*. Note: the return qualifies the result of the block, not a method return.

Signature Qualifier

unary – < *selector, ^ qualifier* >

binary – < *selector qualifier, ^ qualifier* >

keyword – < *firstKeyword: qualifier, secondKeyword: qualifier, ..., ^ qualifier* >

Conjunction Qualifier

(qualifier & qualifier & ...) – all of the qualifiers must apply

This is used to specify protocol that an object must support (via signature qualifiers). For example, not all collections support #add: (e.g., fixed sized collections such as Array). So if a method is, *by design*, intended to accept any collection that can be added to, the argument should be qualified as (hOf Collection & <add: any, ^self>)

Attributes

During static analysis of a method, each variable referenced in the method – whether it is an arg, a temp, an instVar, or a Global. – is modeled by an *Attribute* that is responsible for:

- Knowing what kind of object should occupy the variable. This is specified by the qualifier declared in the design, i.e., the *declared qualifier*..
- Knowing what kind of object occupies the variable as specified by the actions of the code, i.e., the *deduced qualifier*.

- Signalling any improper assignment, i.e., where the deduced qualifier is not acceptable according to the declared qualifier.

There are two exceptions to the rule that every attribute has a declared and a deduced qualifier. Method args have only declared qualifiers because they cannot be set. Method and block temps have only deduced qualifiers because there is no mechanism for declaring their qualifier.

LOCAL STATIC CODE ANALYSIS

Static code analysis of a Smalltalk method is much like the process a programmer takes when reading the code. As we read through the code we mentally parse it, keep track of which objects are assigned to variables, and note whether objects returned from messages sent are assigned to variables or are sent subsequent messages. We view the objects involved in the method more as generic instances of the appropriate class rather than specific instances with specific values of instance variables.

When literal blocks are involved, the task becomes more difficult both for reading the code and for static analysis. Literal blocks are essentially unnamed methods that are defined in-line and invoked by a variety of mechanisms that can be quite subtle. In reading code with blocks, we have to mentally step back and determine when the block will be evaluated and with what arguments, then envision the block being evaluated, perhaps iteratively. In some cases, the block is not evaluated during the execution of the method we are reading. For example, in the case of SortedCollection >> #sortBlock:, the evaluation of the block is completely disconnected from the #sortBlock: message send. Yet we still must mentally analyze the block. We also must be sure we understand the fate of the result of each evaluation of a block (especially in the case of messages such as #collect:, or #inject:into:).

Parse tree analysis

Previous approachs to automated static analysis uses a parse tree derived from the code. Nodes in the parse tree share the ability to answer their “value” in terms of the type (i.e., the qualifier) of the object they would represent at runtime given the qualifiers of objects they depend upon [BI82]. The parse tree is then traversed in a depth first order to obtain the value of all the nodes. In this process, message expressions validate that their arguments are appropriate, variables validate assignments, and finally the value of the whole method (which is its return value) is validated against the design statement of what the method should return.

As is the case when reading code, literal blocks complicate matters for static analysis. Blocks may be assigned to variables that later are sent ‘value’ messages. These blocks are not invoked where they appear in the parse tree and, in fact, they may be invoked more than once. The following example highlights these issues. It has been contrived specifically to challenge a static analyzer. It also is quite a challenge for human analysis.

```
twistedBlockValues
      "<^(hOf String | hOf Integer)>"
      "Example of sending value to literal blocks.
       Note: evaluation order is very different from order of
    appearance and blocks are entered multiple times."

      | firstBlock secondBlock thirdBlock twistBlock |
      twistBlock := [:blkArg1 :blkArg2 | blkArg1 value: blkArg2 value].
      firstBlock := [:myArg | myArg * secondBlock value].
      secondBlock := [self conditionalBlockDepth].
      thirdBlock := [:myArg | myArg printString].
      ^(thirdBlock value: secondBlock value) size > 2
            ifTrue:
                  [twistBlock value: thirdBlock value: secondBlock]
            ifFalse:
                  [twistBlock value: firstBlock value: secondBlock]
```

In this contrived example, the static analysis must not and cannot analyze the blocks when they first appear. At the point they appear in the parse tree, nothing is known about what kinds of objects will occupy the blockArgs. And the analyzer must pass through the blocks each time they are invoked because the types of the blockArgs may (and do) differ when invoked from different places. No one-pass traversal of the parse tree can analyze this code. Yet the design virtual machine approach we describe in this report *can* handle such twisted code. It can also handle blocks involved in methods such as #collect:, #select:, and #inject:into: and some kinds of recursively invoked blocks.

To be sure, the parse tree approach to static analysis probably can be elaborated to handle multiple invocations of the same literal block and perhaps even recursive invocations of literal blocks. But as the analysis departs further from a simple traversal of a parse tree, the parse tree becomes more an impediment to the analysis than an asset. It should be remembered that in the normal compile process, the parse tree is discarded once the bytecodes (including those in block contexts) are generated. This is in part because runtime execution of blocks does not necessarily follow the topology of the parse tree. The value of a parse tree is in generating the bytecodes that are later executed, not in modeling the execution itself.

Design virtual machine analysis

As we will explain, static analysis can be approached in a manner analogous to the way the virtual machine executes compiled Smalltalk code. To understand this approach we must first examine the way Smalltalk virtual machines execute code.

When a Smalltalk virtual machine executes code, three categories of activities occur:

- Interpretation of bytecodes that have been previously compiled from Smalltalk methods
- Handling exceptions, especially doesNotUnderstand, and handling of external events
- The creation and destruction of objects (i.e., memory management)

The visible behavior of Smalltalk code, i.e., behavior that is the subject of static analysis, occurs under the explicit direction of bytecodes. So we ignore memory management and exception handling in the subsequent discussion.

The Smalltalk compiler converts Smalltalk source code to bytecodes. These bytecodes are interpreted by a virtual machine [GR83] [Ing83]. The virtual machine architecture is that of a stack machine. Bytecodes define the pushing of objects from variables (e.g., method args, instance variables, etc.) onto a stack, popping objects from the stack to store them into variables, and sending messages. Message sends pop their arguments and the receiver from the stack, lookup the proper method to invoke, execute that method and push its result onto the stack when the method exits. Each invocation of a method or a block is managed by a MethodContext or BlockContext object that maintains an instruction pointer into its bytecodes and provides private state.

Recall that when we read code to understand its effect, we mentally replace the actual objects with generic stand-ins. Our design virtual machine does something similar; qualifiers stand in for the objects so described. Signatures similarly stand in for methods invoked as a result of a message send. That is, objects and message sends are well described by qualifiers and signatures respectively. The question is how to represent the code itself. Our approach is to represent the code as analogs of bytecodes (hereafter called *execution steps*) that are created from the parse tree in a manner very similar to the generation of bytecodes.

In the design virtual machine approach to static analysis, a specialized virtual machine executes the design, as specified by signatures and qualifiers, in the context of the code as specified by the execution steps. That is, where objects are pushed and popped to and from variables at runtime, qualifiers are pushed and popped to and from attributes by the design virtual machine.

A runtime virtual machine executes message sends by pushing the receiver and args then looking up the method to invoke (beginning in the class of the receiver and working up the inheritance chain). When the invoked method returns, its return value is left on the stack. The runtime evaluation of blockContexts is similar to a message send because blocks are essentially unnamed methods. However, no method look-up is needed

A design virtual machine executes a message send in an analogous manner. Under the direction of execution steps, it begins execution of a method by pushing a self qualifier followed by argument qualifiers. Method look-up differs from the runtime technique in one important way: if the receiving qualifier is an alternatives qualifier, we must analyze the signatures of all of the potentially receivers. Note, however, that we do not need to recursively analyze the method(s) that would be invoked. The design virtual machine need only check that the argument qualifiers on the stack are valid according to the qualifiers in the receiving signature(s). After this process, the return qualifier, determined by the signature(s) of the method(s) that would be invoked, is left on the stack. The evaluation of blocks by the design virtual machine proceeds in a manner similar to a message send.

The following table shows the elements involved in execution of Smalltalk in a typical virtual machine and the corresponding elements in the design virtual machine:

Execution Model	Design Model
Objects	Qualifiers
Message sends	Signatures
Variables	Attributes
Virtual Machine	ExecutionModel
Block/Method Contexts	ExecutionContexts
Bytecodes	ExecutionSteps

Table 1. Correspondence between elements of the execution VM and the design VM

The advantage to be gained over the parse tree approach is that it becomes straightforward to explicitly model literal block contexts with their own state and execution steps and invoke them when appropriate.

Modifications to the compiler's parse tree to generate execution steps is straightforward because the generation of execution steps is very much like the generation of byte codes – a process for which the parse tree is designed. Any optimizations that have been made to the parse tree for good byte code generation are likely to match well the needs of generating execution steps.

A detailed design and a discussion of the operation of this virtual machine is available in a separate IBM Technical Report [BG98b].

DISCUSSION

Analysis of literal blocks

The design virtual machine, which is called the SemanticExecutionModeler or SEM, succeeds quite well in analyzing code containing literal blocks. It not only handles the twisted blocks example but also handles the literal blocks passed as arguments to the familiar control structures such as #do:, #select:, #collect:, and #inject:into:. Analysis of those control blocks does not rely on any special case code in the design VM, it relies on the signatures of those methods in conjunction with the aspects of the collection receiving the message. The following examples illustrate how these signatures interact with the aspect of the receiving collection. Note that iterator methods in collections are virtually the only cases that require signatures as complex as those we discuss below. Signatures for typical user code are usually quite straightforward. Also, note that the default aspect of a Collection is "of:" which refers to the elements of the collection, thus "self aspect" refers to the qualifier of the elements of the collection.

`Collection>>do: <aBlock: [:blockArg1 self aspect,  ^any ], ^ self>` This signature asserts that the #do: method returns self, and that while the analyzer is analyzing the code inside the block, the blockArg is bound to the qualifier of the elements of the block (i.e., `self aspect`). The design execution of the block occurs when the block is popped from the stack to provide the argument for analysis of the #do: message send. The "^any" qualifier within the block qualifier means that no restrictions are placed on the return value of the block.

`Collection>>collect: <aBlock: [:blockArg1 self aspect,  ^any ], ^ iOf mySpecies {of: arg1 value}>` This signature is similar to that of the #do:, except that it asserts that the #collect: method returns a new collection of the same species as the receiver (which is an instance of some subclass of Collection) whose elements are the return value of the block, i.e., arg1 value. That return value is determined by the design execution of the block with the blockArg bound to the aspect qualifier of the receiver.

`Collection>>select: <aBlock: [:blockArg1 self aspect,  ^(true | false) ], ^ hOf mySpecies {of: self aspect}>` This signature is similar to that of the #collect:, except that it asserts that the block, upon analysis, must return true or false, and that the #select: method returns a new collection of the same species as the receiver whose elements are the same as those of the receiver, i.e., self aspect.

`Collection>>detect:ifNone: <aBlock: [:blockArg1 self aspect,  ^(true | false) ], exceptionBlock: [ ^any ], ^ (self aspect | arg2 value)>` This signature asserts that the #detect:ifNone: method returns an alternatives qualifier. The alternatives are the qualifier of the elements of the receiving collection (which itself may be an alternatives qualifier) or the value of the exception block (arg2 value) as determined by static analysis of that block. It also asserts that the detect block, arg1, must return true or false.

`Collection>>inject:into: <initialValue: any, aBinaryBlock: [:blockArg1 arg1, :blockArg2 self aspect,  ^any ], ^ arg2 value>` This signature asserts that the #inject:into: method returns the qualifier determined by evaluating arg2, the binaryBlock. That block is analyzed with its first blockArg bound to the first argument to the method, i.e., the initialValue, and the second blockArg bound to the qualifier of the elements of the receiver.

It should be emphasized that these signatures are associated with the Collection iterator methods themselves, but the assertions they make are about the code in the methods that call the iterators. Consider the following example:

```
iterationExample: input
      "<input: hOf Collection {of: hOf Color}, ^(hOf String | nil)>"
      "A nonsenseical example of literal blocks in iterators.
    The method answers a string if the input collection contains a
    color named red, nil otherwise."

   ^(input collect: [:element | element printString])
        detect: [:string | string = ‘red’] ifNone: [nil]
```

The executionSteps generated from the parse tree will first cause the #collect: message to be “executed” which in turn will cause its literal block argument to be analyzed. The signature of the #collect: method causes ‘element’ to be bound to the aspect of the receiver which the method’s signature tells us is a collection of colors. The signature of the printString method for colors will presumably cause the result of the block to be a string. So the result of the #collect: method analysis will be the qualifier: `hOf Collection {of: hOf String}`. This qualifier will be pushed on the top of the stack which will cause it to be the receiver of the #detect:ifNone: message. Analysis will now proceed through both of the literal block arguments to that method. The signature for #detect:ifNone: asserts that the result of the analysis of the first block must be a boolean which in this case it is; if it weren’t, the analyzer would signal a signature error. The signature for #detect:ifNone: then causes the result of the method to be the alternatives qualifier created by joining the aspect qualifier of the receiver, which is `hOf String`, with the result of the second literal block which is `nil`. So the result of the whole compound statement (also the result of the method) is the qualifier:`(hOf String | nil)` which matches the assertion about the return value in the method’s signature. All is well.

The important point of this discussion is that the design virtual machine knows no more about the workings of these iterator methods and their literal block arguments than the execution VM knows about them at runtime. The SEM merely pushes and pops qualifiers and, when required to evaluate a literal block, creates a new ExecutionContext, which in turn pops its args, steps through its execution steps, and leaves the result of its evaluation on the stack. It is up to the messageSend execution step to create the qualifier for the result of the send from qualifiers available to it from the receiver, arguments, or attributes, as directed by the signature of the method(s) invoked. As we have seen, in the case of these iterators, the signatures are rather complex, but the process is essentially the same as it is with simple signatures.

The sortBlock: method in SortedCollection is an interesting case for analysis of literal blocks. At runtime, the block is executed when the block needs to be sorted, not when the message is sent. In the design static analysis, the block must be analyzed when the #sortBlock: message is sent or not at all (when the sort method itself is analyzed, the code in the literal block is not available). Since the SEM analyzes literal blocks when they are popped from the stack by the message-send execution step, the sort block is, in fact, analyzed in the method that sends the #sortBlock: message. The signature of the SortedCollection>>sortBlock method is:

```
< aTwoArgumentBlock: [:blockArg1 self aspect, :blockArg2 self aspect,
^(true | false)], ^self>
```

In the method that sends the #sortBlock: message, the receiving SortedCollection qualifier usually knows its element's aspect. So the SEM can check the code in the block, and check that the block returns true or false. Of course, it cannot check that the code will actually sort the elements as the programmer intends, only that it will work.

Tools based on the SEM

Real-time static analysis – The most frequent and visible use of the SEM is in conjunction with the real-time quality feedback tool set, called Viceroy and Gypsy, described elsewhere [GB98a, GB98b]. We have incorporated the SEM into these real-time tools so that it is invoked each time a method is browsed or changed. If a mismatch between the signature and the code is found, the user is notified via the Viceroy color coded feedback panels that appear in the augmented VisualAge browsers. These panels also signal results of metrics tests (i.e., is the code too complex? [Bur96]), and the results of checks of the signature syntax. In the case of the results of the SEM analysis, green indicates that the signature matches the code, yellow indicates that one or more warnings were generated in the analysis, and red indicates that a mismatch was found. All of these checks, including the metrics tests, the signature syntax checks, and the SEM analysis, occur quickly enough that the delay in the browser is barely noticeable unless the method is long – a circumstance that is atypical in good code and is discouraged by the complexity metrics.

Design debugging – For visualizing and debugging the design execution, we provide a tool, called the Design Executor or more colloquially, the Stepper. As its name suggests, this tool allows the developer to step through code to follow the "execution" of the design. Its appearance is a bit like that of the standard Smalltalk debugger [see GB98c]. It has a text pane for the code and buttons to control stepping forward or backward through the code (which is highlighted to show the active expression at each step). It also has information panes to show the information one needs in order to understand the analysis as it progresses: the qualifier that results from the highlighted expression, the qualifiers involved in a message send, and the return qualifier from the method. This stepper is quite useful for finding the source of a mismatch between design and code. A button also provides access to the qualifiers of any attributes used by the method. With this tool, it is usually a simple matter to find and correct a problem in the design (or sometimes in the code) by stepping quickly to the error, backing up a step or two and seeing what signatures and qualifiers lead to the problem. The same process – stepping forward and back to see what kinds of objects are involved and how they collaborate – is also very valuable for understanding unfamiliar code where there is no error. This is especially useful for novice Smalltalk programmers who often find it difficult to imagine the execution of a method just by reading the code.

Privacy Enforcement – The SEM also can be configured to enforce rules about who should call a private method. During the SEM analysis, we can detect that a method is private and check if the caller has the right to use that method. Violations are signaled via the Viceroy mechanism.

Batch analysis – At times, e.g., just prior to releasing code to other developers, it is important to check all the code for problems. We therefore provide batch checking tools that analyze a group of methods, e.g., an application, for design/code mismatches. For example the message:

```
   browseAllMethodsWithSEMErrorsInApplication: NAFQualifierModelParser
```

sent to SEMExecutionModel brings up a browser on all of the application's methods in which mismatches are detected.

Search for qualifier usage – Qualifier search tools can find all methods that collaborate with objects specified by a given kind of qualifier. It can be very useful to find all methods that collaborate with an instance of a given class whether that collaboration occurs via an argument, via an attribute such as an instVar, or via the result of a message send.

Producing design diagrams – We have also developed tools for extraction of graphical design diagrams from bodies of code. These tools are described in detail in [GB98d, GB98e]. In brief, there are two sorts of tools. One creates object structure diagrams (OSDs) that include collaboration information in addition to inheritance and attribute information. The collaborators are deduced by running the SEM over all methods in the designated body of code, keeping track of the collaborators discovered. As each collaborator is discovered, we record both the "server" side and the "client" side of the collaboration. These collaboration relationships then appear in the OSD diagram. The second sort of diagram is an augmented object interaction diagram (OID) deduced from a single method. This diagram shows detailed information about each collaboration in the method. Both sorts of diagram are "alive" and interactive. The user can select elements in the diagrams and request additional information about them. In the OID diagram, the user can step into a collaborating method to obtain a new OID for that method.

Effectiveness of the SEM analysis

We propose that this sort of static analysis can be relied upon to ensure that the fine grain design collectively embodied in signatures matches the collective body of code. Thus, it is important that the analysis allow very few false negatives, i.e., cases where the code and the design do not match yet the analysis finds no mismatch. From our experience to date, which involves many person-months of use of the tools in our own development work, we are unaware of any false negatives. That is not to say that situations could not be constructed that would escape detection. It would be foolish to bet against either Murphy's Law or human ingenuity.

False positives are a more common problem. In a static analysis of 2602 methods that have accurate signatures, the SEM warns of possible design/code mismatches in 10.6% of the methods. A careful examination of these methods shows that both the signatures and the code are correct. The most common sources of these false positives are (in approximate order of decreasing frequency of occurrence):

- Implications of guard code – Code inside conditionally executed literal block arguments (i.e., arguments to #ifTrue:, #ifFalse:, #whileTrue:, etc.) commonly relies on some property of collaborators that is not necessarily true outside of that block. That is, guard code in the test condition guarantees that within the block the collaborators can handle the messages sent to them. In most cases, the SEM does not take that guard code into account in its analysis. This is the most frequent source of false positives. Some special-case code has been added to the SEM so that it can make useful deductions from guard code in some of the simplest cases. For example, where an attribute qualifier is checked for being nil:

```
   foo isNil ifFalse: [foo doSomethingNilDoesNotUnderstand].
```

If the qualifier for foo is an alternatives qualifier that includes nil, the SEM deduces that foo is not nil within the block and removes that alternative from consideration. Yet, many conditional tests have implications that the SEM cannot deduce. There is considerable room for improvement in this area. We believe that the majority of these false positives can be eliminated by more intelligent deduction of the implications of the test condition code.

- Abstract protocol – It is common for classes in hierarchies that model complex user or business domains to have one or more abstract superclasses. These abstract classes typically implement methods that are common to the whole hierarchy of their subclasses. And these abstract methods often send messages to self that are implemented only by the concrete subclasses. (Note: it is good practice to put “stub” methods in the abstract superclass for each of the concrete methods implemented in the subclasses, but many programmers do not bother to do so.) The SEM binds ‘self’ within a method to an instance of the class in which the method appears. Thus in analyzing an abstract method, if a message to self is implemented in subclasses but not in the abstract superclass, the message would appear not to be understood. At present this problem is ameliorated by retrying the analysis after making the self qualifier more concrete. The self qualifier is made more concrete by replacing it with an alternatives qualifier containing all the subclasses of the abstract class (we warn the user that this has been done). This solves many of the problems. If there is more than one level of abstract classes, the desired behavior may still not be found. We could do an exhaustive search of subclasses for those that implement the message, but the more clever we get, the more likely we are to leave opportunities for false negatives.
- Assigning nil to attributes that have laissez faire getters – A frequently used design pattern is to provide getters for instVars (or other attributes) that initialize the attribute to some appropriate value if it is nil. A companion practice is to force reinitialization of that attribute by setting it to nil. The signature for the getter and the qualifier for the attribute then appropriately assert that the attribute is not nil, since it will not be if accessed through the getter. When the attribute is set to nil for reinitialization, the SEM detects this as an error. One could change the attribute qualifier to allow nil, but then the SEM would complain about the getter since its signature does not allow nil to be returned, yet the attribute now is allowed to contain nil. Either way, the SEM properly detects an error. This could be fixed by a special qualifier – to be used for attributes or getters only – that formalizes this sort of pattern.

- Unspecified collections – When a new collection is created within a method, populated with elements by code in the method, then used in subsequent code or returned from the method, we must deduce its aspect; there is no declared qualifier to guide us. When a new collection is created in a method, it is given a special aspect qualifier, `unspecified`, that marks the fact that we do not know what it is to contain. As elements are added, the aspect is modified accordingly. There are many ways to alter the elements of a collection. We haven't yet assured that all of them take the unspecified qualifierproperly into account.
- Exception handling blocks – Exception handling, e.g., #when:do: can cause protected code to be resumed with variables fixed, cause a retry, cause the result of the handler block to replace the result of the problem code, or cause an abort. We have made no effort as yet to deal with these issues. Note that exception handling is dealt with as special cases in the runtime VM and therefore probably must be dealt with as special cases in the design VM.
- Use of #perform: – This construct is (and should be) rare. We make no effort to try to analyze the message send, nor is it at all clear that this should be the responsibility of the SEM. The designer/programmer must be willing to specify what selectors are intended to be performed. Perhaps the signature system could be augmented to provide a formal way for this to be done. If so, then the SEM could perhaps enforce those intentions.

False positives aside, most frequently when the analysis signals an error there is a genuine problem. Many errors have been found this way in the authors' code. More often than not, the error is in the method's signature or in signatures of other methods that are used in the code. These errors are usually easy to correct. However they sometimes involve subtle misunderstandings of the implications of design that seems correct. Those cases sometimes turn out to be the tip of the iceberg of a serious design flaw. In addition to design flaws, quite a few bugs have been found in the code. When these errors are not obvious upon inspection, they tend to be the sort that would occur rarely during execution. It is quite gratifying to have these pointed out by the SEM instead of finding them months later after they have caused a serious problem. In general, the experience of working day after day with constant real-time feedback about the correspondence between code and design has reinforced the authors' appreciation of the value of static analysis.

BIBLIOGRAPHY

[BG98a] S. L. Burbeck & S. G. Graham. Using signatures to improve Smalltalk productivity and reuse. IBM Technical Report TR29.3020, 1998.

[BG98b] S. L. Burbeck & S. G. Graham. Implementation of a design virtual machine. IBM Technical Report TR29.3022, 1998.

[BI82] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 133 - 139, 1982.

[Bur96] S. L. Burbeck. Real-time complexity metrics for Smalltalk methods. IBM Systems Journal. Vol. 35, No. 2, pp. 204-226, 1996.

[GB98a] S. G. Graham & S. L. Burbeck. A user's guide to Viceroy functionality. IBM Technical Report TR29.3023, 1998.

[GB98b] S. G. Graham & S. L. Burbeck. A user's guide to Gypsy functionality. IBM Technical Report TR29.3024, 1998.

[GB98c] S. G. Graham & S. L. Burbeck. A user's guide to Monarch functionality. IBM Technical Report TR29.3025, 1998.

[GB98d] S. G. Graham & S. L. Burbeck. Deriving object structure diagrams from Smalltalk code. IBM Technical Report TR29.3027, 1998.

[GB98e] S. G. Graham & S. L. Burbeck. Generating object interaction diagrams from Smalltalk code. IBM Technical Report TR29.3028, 1998.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *17th Annual ACM Symposium on Principles of Programming Languages*, pp. 136 - 150, 1990.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Ing83] Daniel H. H. Ingalls. The evolution of the Smalltalk virtual machine. In *Smalltalk-80 Bits of history words of advice.* Glenn Krasner, ed. pp. 9-28, Addison-Wesley, Reading, MA, 1983.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 187 - 199, 1981.