# What Tracers Are Made of

**Heinz-Dieter Böcker**
GMD-IPSI
Dolivostraße 15
D-6100 Darmstadt
Federal Republic of Germany
boecker@ipsi.darmstadt.gmd.dbp.de

**Jürgen Herczeg**
University of Stuttgart
Department of Computer Science
Herdweg 51, D-7000 Stuttgart 1
Federal Republic of Germany
herczeg@informatik.uni-stuttgart.de

## Abstract

In object-oriented languages like SMALLTALK-80, *browsers* and *inpectors* are used to provide insight into the static world of objects and their relations, *debuggers* are used to inspect and modify states of computation. This article presents a detailed description of TRICK, a basic toolkit to build *tracers*. Tracers can be used to uncover the dynamic properties of SMALLTALK-80 programs. As an example of how the power of this kit may be used by an experienced programmer, we describe the TRACK system, a visual trace construction kit, by means of which trace specifications may be set up through direct manipulation of graphical objects.

## 1 Tracers' TRICK

In object-oriented languages like SMALLTALK-80, *browsers* and *inpectors* are used to provide insight into the static world of objects and their relations, *debuggers* are used to inspect and modify states of computation (cf. [4]). We suggest to build *tracers* that can be used to uncover the dynamic properties of programs. Like browsers and inspectors, tracers live in windows, an arbitrary number of which can coexist concurrently and may cooperate with each other as well as with their cousins: the browsers, inspectors, and debuggers. Also, they are manipulated in similar ways, may be accessed from the other tools, and in turn provide access to these tools.



This paper describes TRICK (tracers' internal construction kit), the low level program interface for the SMALLTALK-80 programmer who wants to build textual or graphical tracers. We will show how it has been successfully used to build TRACK (trace construction kit), which is described in more detail in [2].

Browsers and inspectors obviously can be used to look at the code that implements them. Figure 1 depicts an analogous situation in which the TRACK tracer is applied to some of the components it is built of; the trace shows how a message to be traced is passed through the various subfilters of a *trace filter* which is one of the buildings blocks available within TRICK.

## 2 Tracing Object-Oriented Programs

Standard tracing tools in object-oriented languages (like the one contained in SMALLTALK-80, Version 2.3) are mostly inadequate for other than toy situations. They suffer from basically the same problems that plague the users of tracing tools in ordinary programming languages: for the user it is hard to make them provide just the right amount of information at a level of detail that is just about right for the problem at hand, e.g., to find a bug in a program. A detailed analysis of these deficiencies is contained in the above mentioned paper describing the TRACK system.

Message passing and methods organized in class hierarchies are characteristic constituents of the object-oriented programming paradigm in languages like SMALLTALK-80. The modularity achieved through these mechanisms side-effects the very concept of tracing as known from "ordinary", procedural programming languages. In these languages, tracing refers to some means of monitoring the execution of procedures or

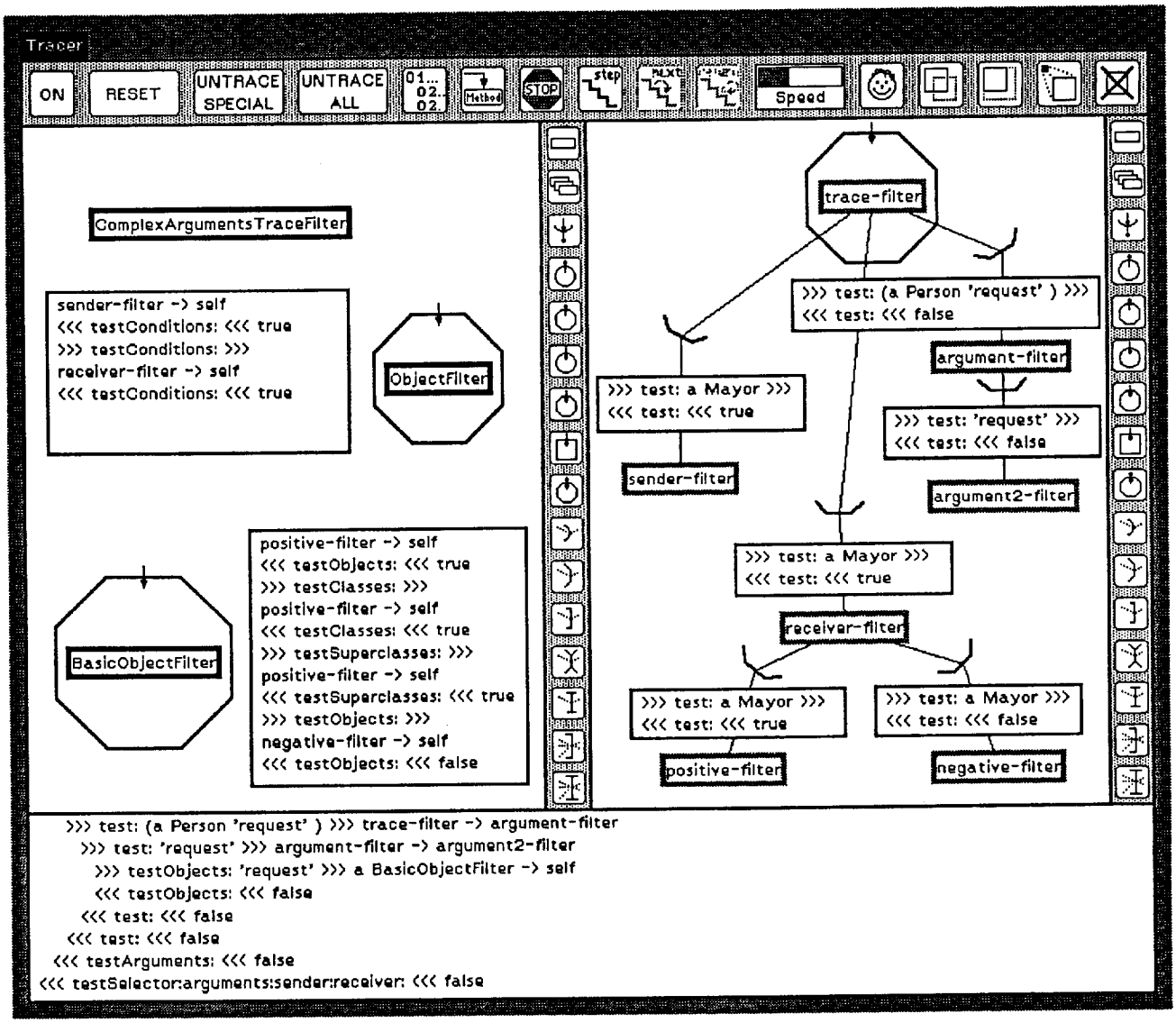


Figure 1: Using the TRACK tracer to trace components of TRICK

functions. In object-oriented languages, tracing may refer to two different, though closely related things: methods or messages. The tracing of methods is roughly equivalent to the tracing of procedures; trace information is produced during (or immediately before or after) the execution of the method or procedure. Alternatively, when focussing on the tracing of messages, we do not necessarily assume the existence of procedural entities that respond to these messages. Tracing messages amounts to spying on information that is flowing *between* objects. A method trace is defined by referring to the recipient of a message or the class of the recipient and its superclasses whereas a message trace does not necessarily refer to any object at all.

Within any tracing task, two phases may be separated:

- *Specification phase*: The program parts, i.e. methods or messages, that are to be traced are specified.
- *Execution phase*: The program is executed and the trace output is generated. This phase may again be separated into two subparts (cf. [5]):
  - *Recording phase*: The trace information is collected, i.e. the program parts to be traced are filtered from all executed program parts.
  - *Animation phase*: The trace information is presented to the user (possibly asynchronously to the recording phase), i.e. arbitrary actions are performed for each program part to be traced.

The TRICK system contains components that may be combined to implement both the specification and execution phase of a trace. By modifying and extending these components or by adding new ones, the specification and animation of a trace can be customized. TRACK (see section 4) is an example of how this may be done by employing graphical interaction and visualization techniques.

Slightly different visualization techniques for tracing object-oriented programs are used by the GRAPH-TRACE system, described by Kleyn and Gingrich [5]. To visualize the flow of control within a program they combine textual protocols with graphical displays of the tree- or net-like structures formed by the calling structures of methods. A similar trace tool is described in [3]. Their system integrates the debugger and a stepping facility to produce diagrams of currently active methods within the context of their containing classes.

# 3 The TRICK System

The TRICK system implements a method trace and provides utilities to implement a message trace. Within TRACK, the basic machinery provided by TRICK has been used to implement a method trace as well as a message trace facility.

TRICK provides a collection of classes which form building blocks to implement user level trace tools. The most important elements of TRICK's architecture are displayed in Figure 2. Below, we will describe the building blocks and show how they work together on the system programming level. Throughout this description we will use the following small program example:

```
bag := Bag with: 1 with: 5 with: 5.
sortColl :=
   SortedCollection with: 9 with: 3 with: 4.
sortColl addAll: bag.
```

Two collections of numbers are created, an unsorted collection (instance of class `Bag`) and a sorted collection (instance of class `SortedCollection`). The elements of the unsorted collection are spliced into the sorted collection. We will trace the communication between the two collections and actions performed within the sorted collection, i.e., the messages sent between both collections, the messages sent from the sorted collection to itself, and the methods invoked. For the sake of simplicity, we will restrict our scrutiny to the methods directly defined in the classes of the two collections; we will not be interested in methods defined in their superclasses, e.g. the class `Object`.

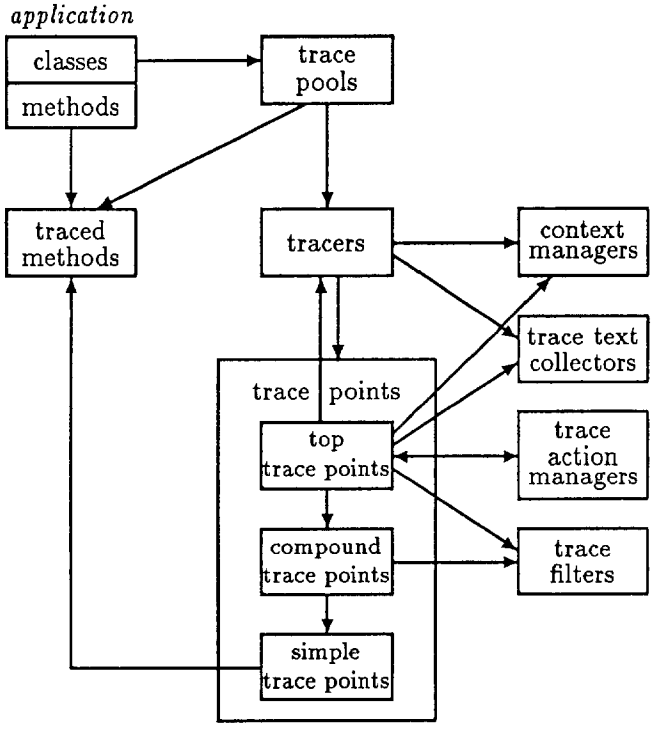


Figure 2: TRICK's basic architecture

## 3.1 Traced Methods and Trace Pools

When deciding about how to implement the foundations of a trace tool, we have to avoid paying performance penalties. We will probably have to accept a small loss in performance for the parts of the program that get traced; the rest of the program, however, should be unaffected. This requirement rules out the possibility of modifying the interpreter and promotes the possibility of modifying the program code, which means that more work is to be done in the specification than in the execution phase.

Obviously there are still alternative ways to implement this kind of tracing mechanism. One method which comes to mind proceeds as follows: define a new method with a name identical to the one of the original method within a new class that is created as a subclass of the original one. From within this method, call the original one via a `super` call. We decided not to use this technique since it would require making all instances and subclasses of the original class into instances and subclasses of the newly created class.

The basic technique used by TRICK to trace methods is illustrated by figure 3. When a method is to be traced, the method dictionary of its class is modified (in the specification phase) in two ways: (A) a new selector entry generated from the original selector is added point-

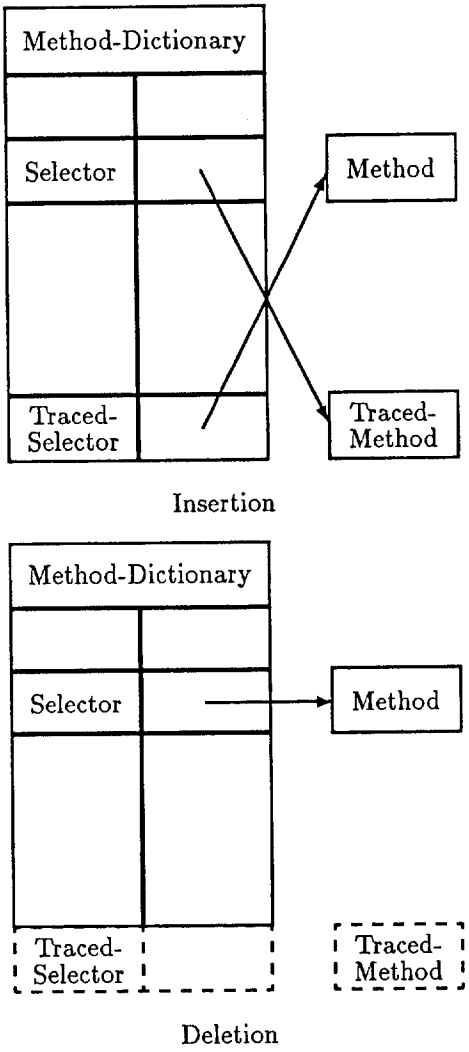


Figure 3: Insertion and deletion of a traced method

ing to the code of the original method, and (B) the original selector is made to point to the *traced method*. This is a piece of code which

1. notifies the class defining the invoked method that this method was called in a certain context (containing sender, receiver, selector, etc.),
2. calls the original method stored away in step (A) and saves the result in a temporary variable,
3. notifies the class of the invoked method that the method called in a certain context is about to finish, and
4. returns the result.

For example, the method dictionary entry of class `SortedCollection` for the method with selector `add:`

```
add: newObject
   | nextIndex |
   self isEmpty ifTrue: :
      [^super addLast: newObject].
   nextIndex := self indexForInserting: newObject.
   self insert: newObject before: nextIndex.
   ^newObject
```

is substituted by two entries for the methods:

```
TRACEDSortedCollectionadd: newObject
   | nextIndex |
   self isEmpty ifTrue:
      [^super addLast: newObject].
   nextIndex := self indexForInserting: newObject.
   self insert: newObject before: nextIndex.
   ^newObject

add: newObject
   | result |
   SortedCollection send: #add: in: thisContext.
   result :=
      self TRACEDSortedCollectionadd: newObject.
   SortedCollection return: #add: in: thisContext.
   ^result
```

It is up to the class containing the traced method how to react to the notifications mentioned. Through methods inherited from the standard class `Behavior` every class knows how to employ *trace pools* for book-keeping of traced methods and how to react to the notification messages. Each class defining methods to be traced has an associated trace pool which maps the traced methods to the *tracers*, instances of arbitrary trace components, in which these methods are to be traced. Thus, each single method may be traced by an arbitrary number of tracers. Conversely, each tracer may trace any number of methods possibly spread over different classes. When a class gets the notification of an invoked method to be traced, it simply passes this information to all corresponding tracers. Each tracer defines whether or not and how the method or the corresponding message are to be traced.

## 3.2 Trace Filters

In common trace situations, not all traced methods introduced in the initial specification phase are of interest to the user any time they are invoked. More typically, the invocation of a specific method is of interest only when called with certain arguments or when being invoked by a message from a specific instance. In TRICK, the methods or messages to be traced are therefore determined in the recording phase (at run time) by *trace filters*. A trace filter is employed to filter a traced

method with respect to its arguments, its result (if already computed), and the sender as well as the receiver of the corresponding message. Since in SMALLTALK-80, message sender and receiver as well as the arguments and the result are objects, in a trace filter each of them is handled by an *object filter*. Filtering aspects for an object passing an object filter are:

- equality with specific objects,
- class membership, or
- arbitrary predicates to be fulfilled by the object.

An object filter is defined as an instance of class `ObjectFilter`. The set of objects being accepted or rejected are specified by the following methods:

```
acceptObject:
acceptInstancesOf:
acceptKindsOf:
rejectObject:
rejectInstancesOf:
rejectKindsOf:
acceptUnderCondition:
```

When no restriction is specified, each object passes the object filter.

A method or message passes a trace filter, when its arguments, result, sender, and receiver pass all object filters and fulfill arbitrary constraints specified for the filter. Trace filters are defined in the specification phase and may be incrementally modified. The object filters are accessible by the methods `senderFilter`, `receiverFilter`, `resultFilter`, or `nthArgFilter:` and may be restricted by the methods of class `ObjectFilter` listed above. Additional constraints for a trace filter are specified by `acceptUnderConstraint:`. From within these constraints the selector, the arguments, the result, the sender, and the receiver can be referenced.

In our example, the trace filters for all messages sent from `bag` to `sortColl` and vice versa may be defined as follows:

```
TFUnsortToSort := TraceFilter new.
TFUnsortToSort senderFilter
   acceptObject: bag.
TFUnsortToSort receiverFilter
   acceptObject: sortColl.

TFSortToUnsort := TraceFilter new.
TFSortToUnsort senderFilter
   acceptObject: sortColl.
TFSortToUnsort receiverFilter
   acceptObject: bag.
```

A trace filter for all messages sent from an instance of class `SortedCollection` to itself is created by:

```
TFSortColl := ComplexTraceFilter new.
TFSortColl receiverFilter
   acceptInstancesOf: SortedCollection.
TFSortColl
   acceptUnderConstraint: 'sender == receiver'
```

## 3.3 Trace Actions

While in traditional trace components the actions performed in the animation phase produce textual output for each traced program part, in TRICK arbitrary *trace actions* may be defined during the specification phase in an object called *trace action manager*. Default actions are textual notifications recorded in *trace text collectors* before and after a method or message has been called. In addition, specific textual output and actions may be specified which, for example, provide a graphical animation of the program (e.g., the "moving ball" within TRACK indicating a "travelling" message) or an invocation or notification of arbitrary programming tools. Specific kinds of actions provided by a trace action manager are *break points* that interrupt program execution before messages to be traced are sent or after their results are returned. A trace action manager is set up by the following methods defined in class `TraceActionManager`:

```
addBeforePrint:
addAfterPrint:
addBeforeAndAfterPrint:
addBeforeAction:
addAfterAction:
addBeforeAndAfterAction:
setBreakBefore
setBreakAfter
```

In our example, for messages sent from the sorted collection to itself we will instantiate a trace action manager, which in addition to the default textual trace output, (A) pretty-prints the receiver object before and after each traced message to see how the message modifies the internals of the collection object and (B) sends a `changed` message to the corresponding tracer to (possibly) update the representation of the objects on the screen in tracers like the graphical tracer of TRACK:

```
TAM := TraceActionManager new.
TAM addBeforeAndAfterPrint: 'receiver'.
TAM addAfterAction: 'tracer changed'.
```

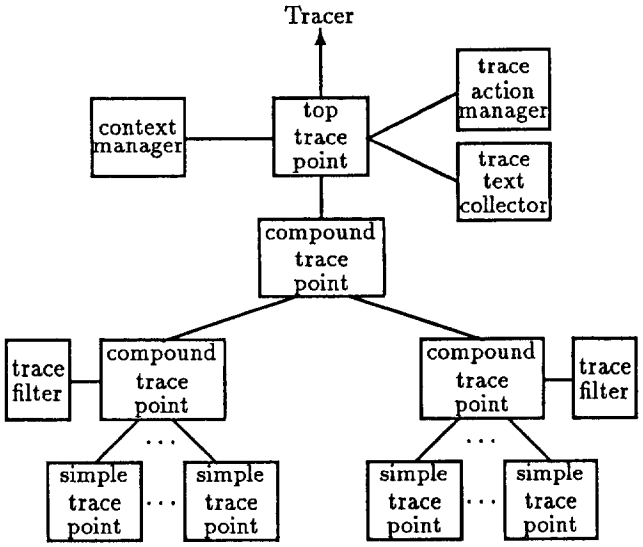


Figure 4: Internal representation of a trace specification

## 3.4 Trace Points

*Trace points* are the main concept used in the internal representation of trace specifications. They tie together the traced methods, trace filters, trace actions, and the tracers in a hierarchical structure, in which it is possible to group sets of traced methods and associate them with a trace filter and specific trace actions. On different levels of a trace point hierarchy, there are three different types of trace points:

- A *simple trace point* corresponds to one traced method; simple trace points form the lowest level of a trace point hierarchy.
- A *compound trace point* groups a set of simple trace points or subordinate compound trace points and associates them with a trace filter.
- A *top trace point* forms the root of the hierarchy and associates all trace points contained with a trace action manager, a trace text collector, a *context manager* which saves the contexts from which traced messages that have not yet returned are sent, and finally a tracer.

Thus a trace point hierarchy for each traced method associated with its simple trace points defines *when* the method or the corresponding message is to be traced and *what actions* are to be performed. The method is traced if it passes all trace filters (cf. figure 1) following the path of the hierarchy starting at the corresponding simple trace point up to the top trace point.

The trace point hierarchy for a trace of all messages sent between the two collections `bag` and `sortColl` of our example may be set up as follows (cf. figure 4):

```
CTPUnsortToSort := CompoundTracePoint new.
(SortedCollection
   traceableMethodsCalledFrom: Bag) do:
   [:selector | CTPUnsortToSort addTracePoint:
                (SimpleTracePoint new
                   method: selector
                   class: SortedCollection)].
CTPUnsortToSort traceFilter: TFUnsortToSort.

CTPSortToUnsort := CompoundTracePoint new.
(Bag traceableMethodsCalledFrom:
   SortedCollection) do:
   [:selector | CTPSortToUnsort addTracePoint:
                (SimpleTracePoint new
                   method: selector
                   class: Bag)].
CTPSortToUnsort traceFilter: TFSortToUnsort.

CTP := CompoundTracePoint new.
CTP addTracePoint: CTPUnsortToSort.
CTP addTracePoint: CTPSortToUnsort.

TTPBetwColls := TopTracePoint new.
TTPBetwColls subTracePoint: CTP.
```

The trace for all messages sent from a sorted collection to itself is specified by the following operations:

```
CTPSort := CompoundTracePoint new.
(SortedCollection traceableMethodsCalledFrom:
   SortedCollection) do:
   [:selector | CTPSort addTracePoint:
                (SimpleTracePoint new
                   method: selector
                   class: SortedCollection)].
CTPSort traceFilter: TFSortColl.

TTPSortColl := TopTracePoint new.
TTPSortColl subTracePoint: CTPSort.
TTPSortColl traceActionManager: TAM
```

## 3.5 Tracers

A *tracer* implements the specification and execution of a method and/or message trace. It implements the construction and modification of trace point hierarchies, makes sure that each method represented by a simple trace point is converted to a traced method which in turn addresses the tracer by the trace pool associated with the corresponding class. In the execution phase, a tracer redirects the notifications from executed traced methods to the corresponding trace points where the methods (or messages) to be traced are filtered out and the trace actions are performed. Moreover, a tracer may visualize trace specifications and, for example, implement a graphically animated trace. Since the trace points are not directly addressed by the traced methods, the whole trace specification of a tracer may be

easily activated and deactivated to temporarily silence a tracer without removing trace points, and mechanisms like a stepping facility may be implemented.

A tracer specifies how to trace a part of a program. Multiple tracers may be used to trace different program parts, e.g., to separately trace the methods contained in separate categories of a class or methods on different abstraction levels of the application. Different tracer specifications and different kinds of tracers, possibly referring to the same methods, may coexist simultaneously.

A simple tracer may implement a textual trace printed to a scrollable text window. In our example, this tracer is set up in the following way:

```
SimpleTracer := Tracer new.
SimpleTracer addTracePoint: TTPBetwColls.
SimpleTracer addTracePoint: TTPSortColl.
SimpleTracer open.
```

Figure 5 shows part of the textual trace output of this simple tracer in the trace text collector for messages from the sorted collection to itself, when the unsorted collection is spliced in.

A message that has been sent is marked with >>> and identified by its selector, arguments, sender, and receiver. The result returned by a message is printed together with the corresponding message selector marked by <<<. Hierarchical dependencies between message calls are represented by appropriate indentation. Additional trace output like the message receiver in the trace example (`receiver = ...`) and notifications for performed trace actions (`tracer changed ... done`) are printed below the default entries.

When a method or message is traced with TRICK, the following actions are performed within the different components between the invocation of the traced method and the evaluation of the trace actions:

1. The class in which the method is defined is notified that the method will be invoked, i.e. the message has been sent.
2. All tracers possibly tracing the method or message are notified via the trace pool associated with the class.
3. By the trace filters, the *active* trace points are filtered out of all trace points of the tracers referring to the traced method.
4. The trace actions before method invocation for the active trace points are executed.
5. The original method is invoked.

```
...
>>> reSort >>> a SortedCollection -> self
receiver = SortedCollection (3 4 9 5 5 1 )
 >>> sort: 1 to: 6 >>> a SortedCollection -> self
 receiver = SortedCollection (3 4 9 5 5 1 )
  >>> swap: 1 with: 6 >>> a SortedCollection -> self
  receiver = SortedCollection (3 4 9 5 5 1 )
  <<< swap:with: <<< a SortedCollection
  receiver = SortedCollection (1 4 9 5 5 3 )
  tracer changed ... done
  >>> swap: 6 with: 3 >>> a SortedCollection -> self
  receiver = SortedCollection (1 4 9 5 5 3 )
  <<< swap:with: <<< a SortedCollection
  receiver = SortedCollection (1 4 3 5 5 9 )
  tracer changed ... done
  >>> sort: 1 to: 1 >>> a SortedCollection -> self
  receiver = SortedCollection (1 4 3 5 5 9 )
  <<< sort:to: <<< a SortedCollection
  receiver = SortedCollection (1 4 3 5 5 9 )
  tracer changed ... done
  >>> sort: 2 to: 6 >>> a SortedCollection -> self
  receiver = SortedCollection (1 4 3 5 5 9 )
   >>> sort: 2 to: 3 >>> a SortedCollection -> self
   receiver = SortedCollection (1 4 3 5 5 9 )
    >>> swap: 2 with: 3 >>> a SortedCollection -> self
    receiver = SortedCollection (1 4 3 5 5 9 )
    <<< swap:with: <<< a SortedCollection
    receiver = SortedCollection (1 3 4 5 5 9 )
    tracer changed ... done
   <<< sort:to: <<< a SortedCollection
   receiver = SortedCollection (1 3 4 5 5 9 )
   tracer changed ... done
   >>> sort: 4 to: 6 >>> a SortedCollection -> self
   receiver = SortedCollection (1 3 4 5 5 9 )
   <<< sort:to: <<< a SortedCollection
   receiver = SortedCollection (1 3 4 5 5 9 )
   tracer changed ... done
  <<< sort:to: <<< a SortedCollection
  receiver = SortedCollection (1 3 4 5 5 9 )
  tracer changed ... done
 <<< sort:to: <<< a SortedCollection
 receiver = SortedCollection (1 3 4 5 5 9 )
 tracer changed ... done
<<< reSort <<< a SortedCollection
receiver = SortedCollection (1 3 4 5 5 9 )
tracer changed ... done
...
```

Figure 5: Textual trace output

6. The method defining class is notified that the method has been evaluated, i.e. the message will return a result.
7. All tracers possibly tracing the method or message are notified.
8. The active trace points are filtered out, the result of the message is included in the filtering process.
9. The trace actions after method invocation are executed for the active trace points.

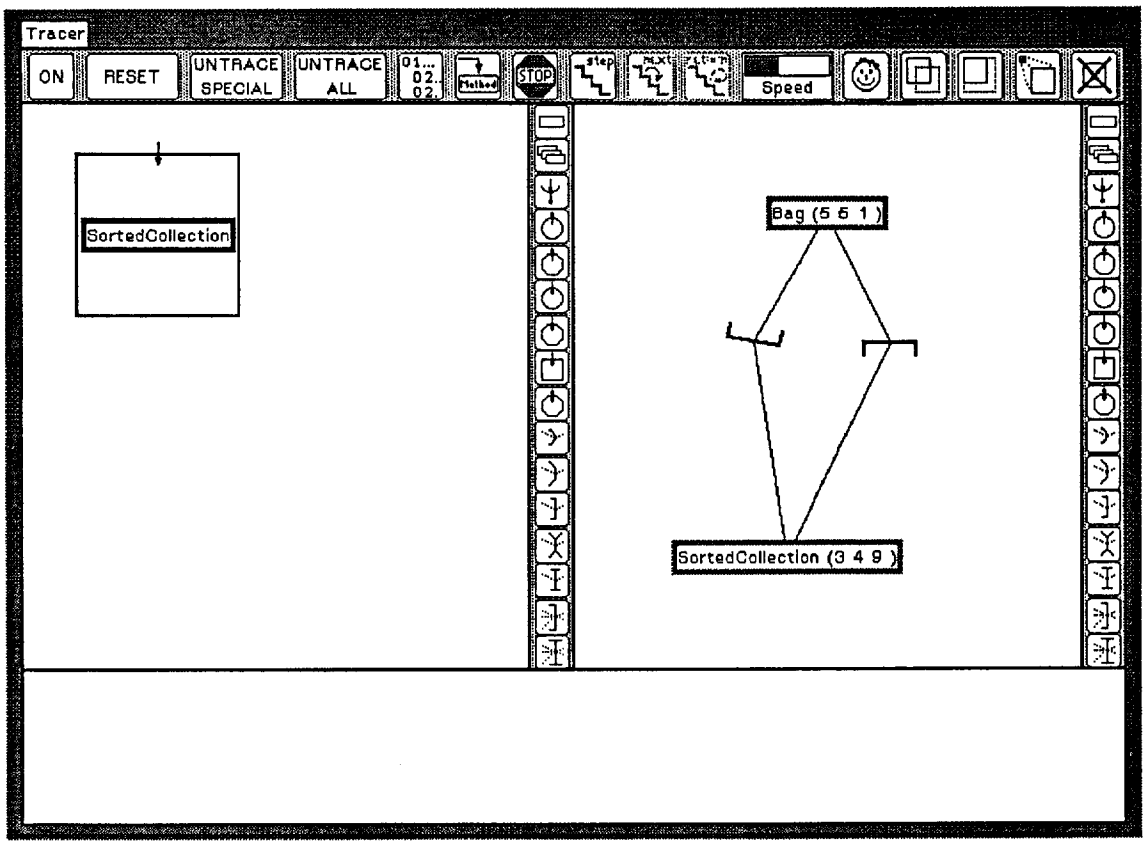


Figure 6: Trace set up with TRACK before splicing in the collection

# 4 TRACK – A Graphical Tracer

The TRACK system implements a tracer in which both trace specifications and trace output are visualized graphically [2]. A message trace is specified by setting up obstacles between or around graphically visualized objects (classes and instances) to be affected by the trace (cf. figure 1), e.g., to trace all messages between two instances, a "hurdle" is placed between the icons representing them, or to trace all messages to instances of a specific class, a "fence" is built up around the icon representing this class. Different types of obstacles that represent different trace specifications are predefined. There are round shaped obstacles which trace exactly one type of message and obstacles having rectangular shape tracing all messages. The type of obstacle is chosen from a graphical menu, and the position is determined interactively by moving the mouse under continuous feedback indicating what the affected objects are. When an obstacle is set up by the user, the corresponding internal trace point hierarchy with appropriate trace filters and default trace actions is automatically built up. The trace filters and trace actions may be incrementally modified via dynamic menus and forms.

When traced methods are invoked, the corresponding messages and methods are textually recorded in windows associated with the trace text collectors. There may be textual trace windows for each obstacle and there is a global trace window for all messages and methods traced by the tracer. Further, each traced message is graphically animated by a little ball moving from the message's sender to its receiver, following the line connecting the two, eventually crossing the obstacle.

Figures 6 and 7 display a graphical tracer of TRACK tracing the example used throughout this article. The graphical tracer of figure 6 contains visualized representations for all instances of class `SortedCollection` in the left part of the trace window and for the two specific instances of `Bag` and `SortedCollection` in the right part of the window. Obstacles are set up between or around these objects that trace all messages between the `Bag` and the `SortedCollection` (the two hurdles in the instances' trace window) and all messages sent from a sorted collection to itself (a square fence around the icon representing all instances of class `SortedCollection`). The additonal filter constraint `sender == receiver` and the `tracer changed` – action are specified for the obstacle in the class trace window via a menu attached to the obstacle and by entering the appropriate expressions. When the `sortColl addAll: bag` expression is

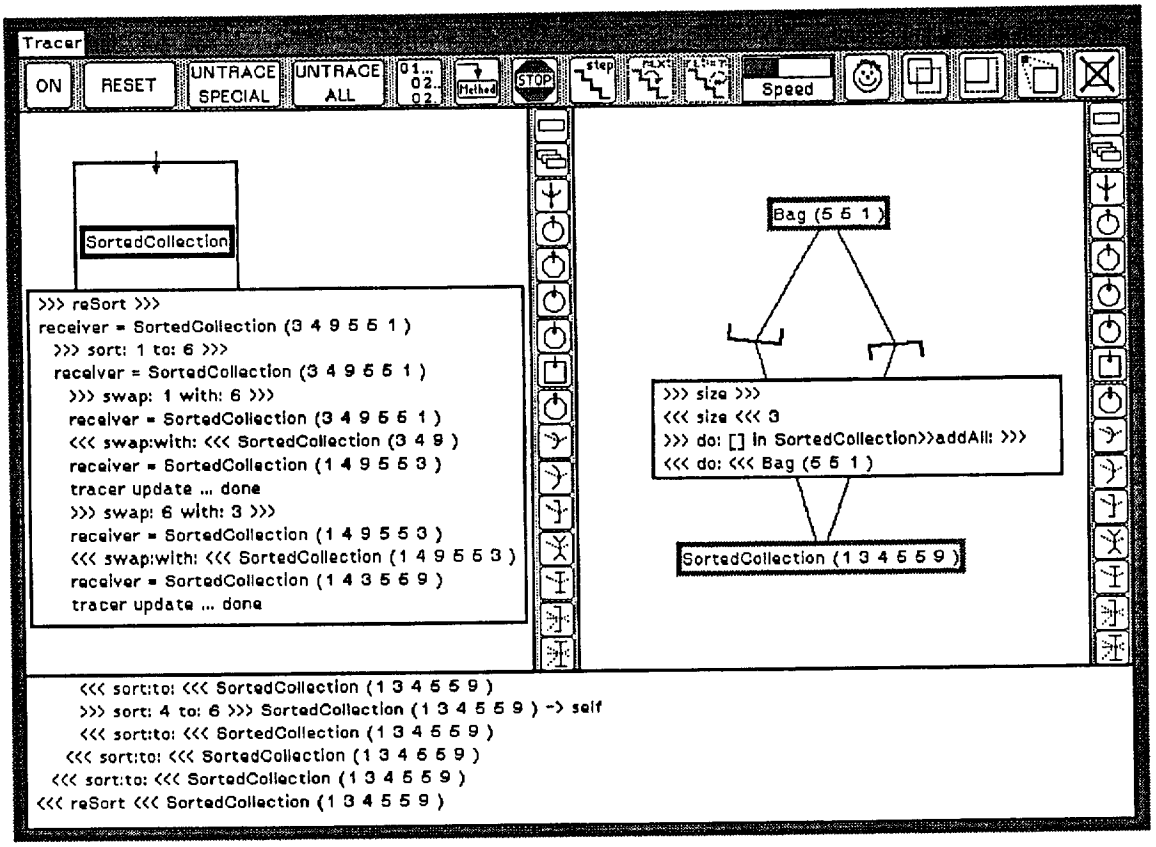


Figure 7: Graphical tracer after splicing in the collection

evaluated, an animation starts which visualizes the messages sent between the objects and shows how these objects are changed in response to the messages: For each traced message a ball moves from the sender to its receiver. When it crosses the obstacle, a textual notification of the message is printed in the global trace text window in the lower part of the tracer and in the local trace text window of the obstacle (cf. figure 7). Local trace text windows may be opened by clicking onto the corresponding obstacles. Moreover, by the `tracer changed` action specified for messages from a sorted collection to itself, the textual identifications in the object icons in the trace windows are updated. Thus, after each message crossing the obstacle surrounding the class `SortedCollection`, the numbers currently contained in the instance of `SortedCollection` and their order are displayed. Figure 7 shows the graphical tracer after all numbers of the `Bag` have been inserted into the `SortedCollection` at the correct position.

For each obstacle, the user may set a break point which causes the program execution to be interrupted when an animated message crosses the obstacle. The message may be inspected and the program execution continued by clicking on the ball visualizing the interrupted message.

By pressing the appropriate buttons and switches in the top menu of a graphical tracer, the following utilities are invoked:

- All messages and methods in the textual trace output may be marked by a unique number to identify corresponding "before" and "after" entries in the different trace text windows.
- In addition to the traced messages, the invoked methods determined by selector and class are printed in the global trace text window when the corresponding messages are received (method trace).
- The program execution may be interrupted by the `STOP` button whenever an animated message crosses an obstacle just as if a break point would have been encountered.
- The program may be executed in a *stepping mode*. In each step, one traced message is sent. The user requests to see more traced messages sent within the corresponding method invocation or to jump without further trace information to the point where the current message returns its result.
- The speed of the program animation may be adjusted by a graphical gauge.

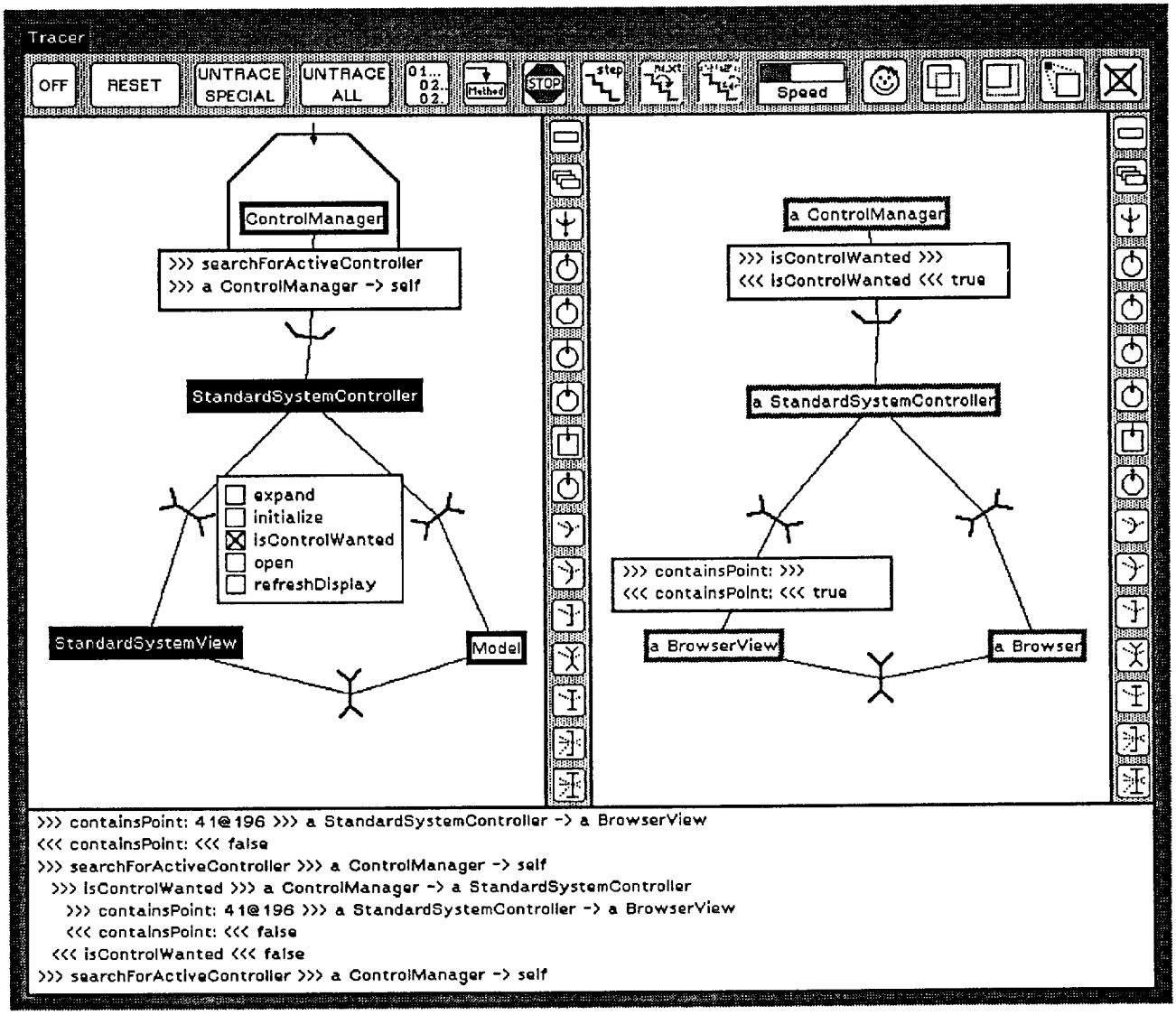


Figure 8: Animating the MVC-paradigm with TRACK

Figure 8 shows how TRACK may be used to animate and analyze even complex dependencies within SMALLTALK-80 such as the Model-View-Controller user interface paradigm [6], which is much more difficult to understand from inspecting the static structure of the objects involved, let alone from the program code.

Tracers like TRACK are not just tracing tools. They may also be used in a more general sense as *execution browsers*. Traditional browsers are used to explore the static properties of code or other symbolic structures. By constructing (or spawning) tracers, that are similar to each other and by interactively modifying them the dynamic behavior of a program may be explored in similar ways [1]: the grain size of observation may change from coarse to detailed, different sections of a SMALLTALK-80 program may be included in different tracers, the speed of execution may be adjusted, etc.

# 5 Integration of TRICK and TRACK with SMALLTALK-80

The TRICK system as well as the TRACK tracer have been fully integrated into the SMALLTALK-80 programming environment. Making it possible to access a tracer from the standard programming tools of SMALLTALK-80 like browsers, inspectors, and debuggers and, conversely, making these tools accessible by a tracer is crucial with respect to usability. In a graphical tracer, for example, visualized classes and instances may be inserted and inspected with standard class browsers and inspectors, respectively; traced messages may be inspected by debuggers. It is possible to specify a textual trace of specific methods or all methods of a class from a method or class browser.

To integrate this trace component with SMALLTALK-80, the following major problems had to be solved:

- The existence of extended methods introduced to implement traced methods and messages had to be hidden from the programmer; the browser had to be modified to show the original methods and, in case the programmer modifies them, propagate the changes to the traced methods.

- Deleting and adding methods or classes in an application that is being traced causes several consistency problems, e.g. what happens when a class is removed which has traced methods and is visualized in a graphical tracer, or what should happen when a method is added to a class for which the user specified all methods to be traced? To solve these problems, each tracer is notified about the creation or deletion of classes and methods to perform appropriate update actions that keep each trace specification consistent with the rest of the system, e.g. when a class is removed it is also removed from all graphical tracers, or when a method is added, the corresponding traced method is automatically created if any trace specification implicitly addresses this method.

- To let the user also trace methods that are sent within the trace component itself (methods of which the user generally is not aware), a special locking mechanism has been introduced to avoid infinite recursion during the internal trace process, see for example figure 1 where the messages sent to a trace filter are themselves filtered by a trace filter.

# 6 Conclusion

In this paper, we have introduced the concept of *tracers* and suggested them to be one of the core ingredients of sophisticated programming environments. We have also tried to give some evidence that the TRICK system described in this paper provides the right set of primitives to implement tracers by describing how it was used to built TRACK, a graphical tracer.

However, the potential of trace tools like TRACK and TRICK is by no means limited to pure tracing. They may easily be employed for the implementation of general algorithm animation tools by combining them with other tools, e.g. visual browsers and inspectors.

# References


[1] Heinz Dieter Böcker and Jürgen Herczeg. Browsing Through Program Execution. In *Proceedings of INTERACT'90, IFIP Conference on Human-Computer Interaction*. IFIP, 1990, forthcoming.

[2] Heinz Dieter Böcker and Jürgen Herczeg. TRACK — A Trace Construction Kit. In *CHI-90, Human Factors in Computing Systems Conference Proceedings*, pages 415–422. ACM SIGCHI/HFS, 1990.

[3] Ward Cunningham and Kent Beck. A Diagram for Object-Oriented Programs. In N. Meyrowitz, editor, *OOPSLA '86 Proceedings*, pages 361–367, September 1986.

[4] A. Goldberg, editor. *SMALLTALK-80, The Interactive Programming Environment*. Addison-Wesley, Reading, Ma., 1984.

[5] Michael F. Kleyn and Paul C. Gingrich. GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views. In N. Meyrowitz, editor, *OOPSLA '88 Proceedings*, pages 191–205. Schlumberger-Doll Research, November 1988.

[6] G.E. Krasner and S.T. Pope. A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80. ParcPlace Systems, 1988.