# State Oriented Program Comprehension in legacy distributed telecommunication systems*

Stéphane S. Somé, Timothy C. Lethbridge
School of Information Technology and Engineering (SITE)
150 Louis Pasteur, University of Ottawa, Canada
ssome@csi.uottawa.ca

## Abstract

*Distributed telecommunication systems consist of several programs that spawn independent but collaborating processes during their execution. These processes are implementations of state-based systems.*

*Understanding programs is a pre-requisite for the effective maintenance of a software system. Program comprehension aims at helping software maintainers understand programs by acquiring knowledge about them [15]. One of the approaches for program comprehension, is to provide maintainers with browsing tools that highlight elements in source code, show their relationships and infer high-level abstractions.*

*This paper discusses how to consider the state-based dimension of distributed telecommunication systems in the design of program comprehension facilities.*

**Keywords :** program comprehension, state, state transition, distributed system, telecommunication system.
**Research paper**

## 1. Introduction

Program comprehension tools helps software maintainers get an abstract view of a software system. A key task is highlighting elements in the system source code and showing their relationships. The abstract view assists in navigating through the software code, understanding how it works, understanding its design and assessing the impact of changes.

Program understanding tools are particularly critical for the maintenance of complex software systems. Our research targets such systems [9, 10]. We are particularly working with a large distributed PBX system developed by Mitel corporation.

*This work is supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER).

Software used in PBX systems, because of their size and distributed nature, constitute a class of complex software difficult to understand and maintain. The software of the PBX system we are working with consists of about 1.5 million lines of code in 4000 source files. The software engineers who work with this system regularly add new features, fix problems and re-engineer portions of it.

There are several program comprehension tools in the research and commercial domains [16, 13, 12, 5]. These tools allow program understanding about elements directly derived from programming languages such as routines, data types, data variables, classes, or higher abstractions built from such elements. Our research also produced a tool called TKSEE [4] with the similar code exploration features. TKSEE is already used for the maintenance of the PBX system used in this paper.

Distributed systems are characterized by the involvement of several independent programs that may reside at different physical locations. These programs are generally modeled upon finite state machines [17]. Finite state machines [2, 3] are characterized by the concepts of `state` and `state transition`. A state describes a certain situation and a state transition a change from one state to another. Transitions between states are generally provoked by `specific events`. Because the source code is a mapping of states and transitions using some underlying programming constructs, states and transitions are important concepts for understanding the *logic* and *structure* of state based systems.

In this paper we present an approach to extract information about states in a large distributed telecommunication system in order to design state-oriented code exploration facilities. The objective of the research is to enhance the program exploration tool TKSEE with state-based browsing facilities. The paper focus on a particular system. However, we believe that the underlying ideas are applicable to many other distributed systems and that the approach can be generalized.

The paper is organized as follows. In the next section, we describe the model of the PBX system used for experi-

mentation. Section 3 then presents our approach to support state based program comprehension based on this particular model. Finally section 4 outlines our ongoing work and concludes the paper.

## 2. State model in a PBX system

A PBX is a key component in a telecommunication system. It is an embedded hardware/software system in charge of establishing and managing communications between external entities. The PBX system we are working with includes a Main Controller, a set of Peripheral Controllers and a `process` based operating system. The operation of the PBX consists of several concurrent collaborating processes. Each process is an incarnation of a program.

### 2.1. Call Processes

The system uses a concept of `call processes` in direct relation with its telecommunication nature. A call process is a separate process executing a portion of a call. Typically several call processes will collaborate by sending messages each to the other in order to achieve a communication between external parties.

Figure 2 shows call processes involved in a two party call. These call processes control the progression of the call by exchanging messages and communicating with the physical devices through their Peripheral Controllers.

The communication scenario in the system is as follows : When an activity is sensed on a peripheral device (example hook off a telephone handset), the Peripheral Controller sends a message describing the activity and the state of the device to the Main Controller. A Message Subsystem of the Operating System then routes the message to the appropriate call process depending on the device type. The call processing may then lead to various interactions with other call processes within the Main Controller. Ultimately a call process may address a message to another Peripheral Controller which interprets it and performs some appropriate action on the controlled device.

### 2.2. Call states

Any process has an operating system state. This state may be dead, ready, running, waiting or stopped. The operating system state is manipulated by the operating system and serves for purposes such as scheduling. We are not interested in these operating system states. There is another kind of state for call Processes : `call states`.

Call states are maintained by call processes to describe the status of the calls they are managing. At an abstract level, the software in the PBX is as shown in Figure 3, composed of interacting state transition systems.

Call process behavior depend on their state. As an example, when a call process receives a message from a Peripheral Controller (in reaction to some activity in the pheripheral), it interprets it according to the state of the device that sent the message, and its own call state.

### 2.3. Mapping states to code

The call management software in the PBX is a mapping of state transition systems. In the following, we describe how this mapping is done.

A process' `call state` is a combination of `call state values`. The list of possible call state values are defined by `call state types` ; enumerated types whose fields are call state values.

Figure 1 shows an example of a call state type definition. A naming convention is used to distinguish these types to other enumerated types in the particular system we are studying. Call state types have the word “state” in their name.

```
typedef enum devxx_call_states_t {
        devxx_cs_nil,
        devxx_cs_idle,
        devxx_cs_dialing,
        devxx_cs_route_determined,
        devxx_cs_failed_call,
        devxx_cs_seized,
        devxx_cs_connected,
        devxx_cs_unavailable,
        devxx_cs_last_call_state
    } devxx_call_states;
```

**Figure 1. Call state type definition. This is an enumerated type that defines a set of call state values as its fields.**

A call state value indicates one of many dimensions of the status of call processing. As an example, the status of each of the devices involved in a call will correspond to a state value.

State values are held by `state variables` in the source code as shown in Figure 4.

These values are used to control call processes behavior as shown in Figure 5.

State variables also include some *indirect* instances of call state types. These are instances of record types whose fields include direct or indirect instances of call state types. Figure 6 shows an example of such a record type. Instances of devxx_call_status are state variables because the field callstate hold state values.

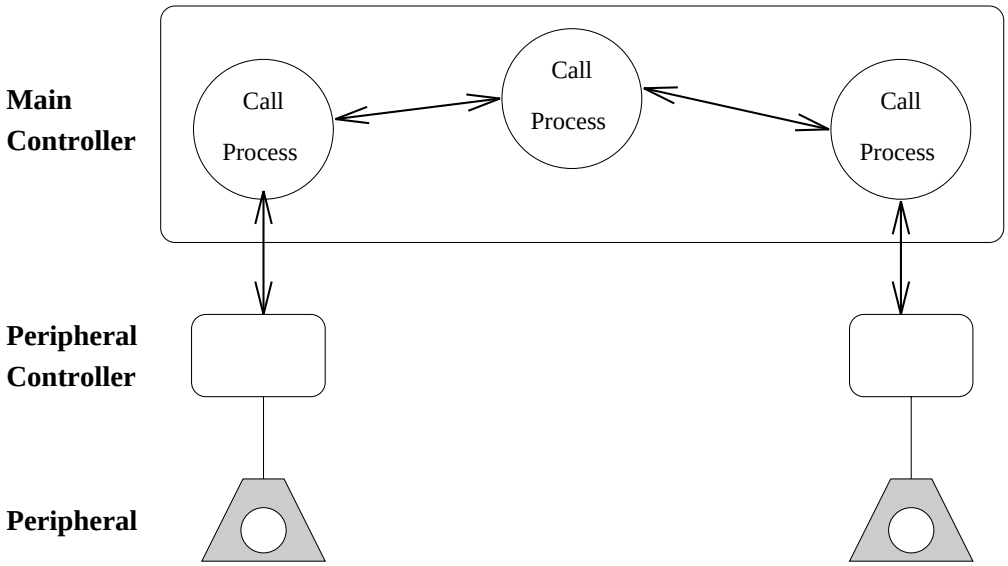


**Figure 2. Interaction between call processes.**

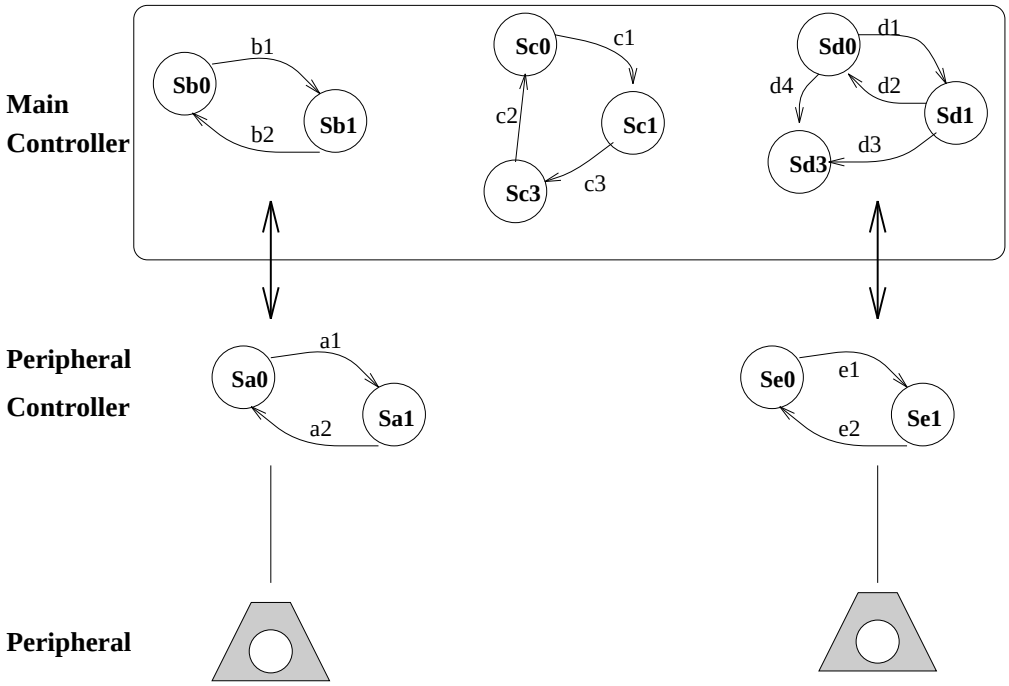


**Figure 3. Call state model.**

```
. . .
devxx_call_states devxx_state;
. . .
if    (party.status == _dialing &&
      party.call_state != ringing)
      devxx_state = devxx_cs_unavailable;
else
      devxx_state = devxx_cs_idle;
. . .
```

**Figure 4. Setting process state values. In this example one of the state values of the invoking process is set according to some conditions. The state variable used is devxx_state.**

```
. . .
devxx_call_states devxx_state;
. . .
switch (devxx_state) {
      case devxx_cs_idle:
            . . .
            break;
      case devxx_cs_dialing:
            . . .
            break;
      case devxx_cs_route_determined:
            . . .
            . . .
      default :
            . . .
};
. . .
```

**Figure 5. Control of a process behavior using state values.**

```
typedef struct devxx_call_status_t {
            . . .
            int ref;
            devxx_call_states callstate;
            devxx_call_types calltype;
            . . .
      } devxx_call_status;
```

**Figure 6. Record type which field is a state type instance.**

## 3. State based program comprehension

### 3.1. Objective

Our objective is to develop new program understanding facilities to help software maintainers explore source code with a state model perspective.

A state model gives a high level view of the behavior of a system. It reflects in the system source code where `low-level` programming elements such as routines and data objects are organized and related to each other according to the model. Having the state model of a system in mind therefore gives additional information to a software engineer who want to understand and maintain it.

As shown in Figure 5, `state values` are used to control the behavior of processes using if and/or case-like statements. The exploration includes showing the source code controlled by given `states` or `state values`. As an example, a maintainer may want to know:

- the state values that are set at the same time,
- the routines called in a given state or when a given state value is set,
- the states that can be reached from a given state,
- the state variables used in a program,
- etc.

State based exploration does not supersede traditional program browsing. Queries about states are supposed to be used in conjunction with program browsing inquiries based on relationships between elements in the source code.

The state-based exploration facilities we are developing here will be added to TKSEE a program browsing tool. TKSEE uses a database of software objects (e.g. routines, types, variables) and their relationships (e.g. uses, definitions) obtained by parsing the source code. Adding state-based program understanding facilities consist of adding new objects and relationships to the database. However, a first step before state extraction is to group source code according to processes.

### 3.2. Finding source code belonging to processes

We are working with a PBX system made of several call processes. The software in the system is large (1.5 million line of code in about 4000 source files).

Each call process has a corresponding main file where its main function is defined. But several processes share other source code scattered in different files.

It is necessary to delimit the source code accessed by a call process in order to find its states. The approach starts

from each call process program's main function and follows the call graph involved in the Process.

For each process P, we determine `routines(P)` the set of routines called by P, `types(P)` the set of data types manipulated in P and `variables(P)` the set of global variables manipulated in P. These sets are then used to find all the files related to the process P by considering all the source files where the elements in `routines(P)`, `types(P)` and `variables(P)` are defined.

These sets make it possible to explore the system with a process perspective since one can precisely know what source code is pertinent to a given call process. Another useful capability is to know the source code exclusive to a call process; this source code (routines, types, variables) therefore can be modified without any influence on other processes. On the other hand, it is also possible to know source code shared by several processes and therefore be able to evaluate the impacts of changes made in that code.

### 3.3. Extracting a complete state model

A call state is a combination of call state values assigned to state variables at some point in the source code executed by a call process.

Figure 4 shows a typical example with the setting of variable `devxx_state`. The possible values of `devxx_state` are enumerated by the call state type `devxx_call_states` described in Figure 1.

More formally, a state S is determined by a set of state values $\{sv_1, \cdots, sv_n\}$ such that each $sv_i \in$ `state_values(P)` the set of all the state values used by the process P.

A process is in a `call state` S = $\{sv_1, \cdots, sv_n\}$, at some point of its execution, if there are n variables of different `call state types` set to respectively $sv_1, \cdots, sv_n$.

Given a call process P, we can determine `states(P)`, the set of all the `call states` of P as follow.

1. Identify state types.

   In the general case there is no "explicit" way to distinguish call state types from other enumerated types. However in the PBX system we are working with, there is a naming convention to distinguish state types. Expert knowledge may be needed in the general case.

2. Find `state_values(P)`, all the state values used by P.

   `State_values(P)` is the set of all state values refered in the main program of P and in all the routines in `routines(P)`, all the routines called directly or indirectly from P main program.

3. Let $state\_values(P)^2$ be the power-set of `state_values(P)`.

4. `states(P)` is the set of all elements $\{sv_1, \cdots, sv_n\}$ in $state\_values(P)^2$ such that for each $sv_i$ there is no $sv_j$ such that type($sv_i$) = type($sv_j$).

As an example, imagine a process P which refers to the state values $\{sv_{11}, sv_{12}, sv_{21}, sv_{31}\}$ and suppose that:

- $sv_{11}$, and $sv_{12}$ are defined by the `call state type` $st_1$,
- $sv_{21}$ is defined by the `call state type` $st_2$ and
- $sv_{31}$ is defined by the `call state type` $st_3$.

`State(P)` is the set $\{\emptyset, \{sv_{11}\}, \{sv_{12}\}, \{sv_{21}\}, \{sv_{31}\}, \{sv_{11}, sv_{21}\}, \{sv_{11}, sv_{31}\}, \{sv_{12}, sv_{21}\}, \{sv_{12}, sv_{31}\}, \{sv_{21}, sv_{31}\}, \{sv_{11}, sv_{21}, sv_{31}\}, \{sv_{12}, sv_{21}, sv_{31}\}\}$.

A `state transition` denotes a change from a state to another one in a process. A `state transition` is reflected in the source code as the modification of some `state variables` which are set to new `state values`. Figure 4 shows possible state transitions with the setting of the `state variable` dev_state.

Formally, there is a state transition from a state $s$ to a state $s'$ in a process P if:

- $s$ corresponds to a set of states values $SV$ and $s'$ corresponds to a set of states values $SV'$,
- at some point in the process P source code, the set of `state values` known is $SV$,
- there is a statement such that the set of `state values` becomes $SV'$.

$SV'$ may be obtained by adding a new `state value` to $SV$ or by replacing an old value by a new value of the same `state type`. The first case corresponds to the setting of a `state variable` that doesn't have a value while the second case corresponds to the resetting of a `state variable`.

A `state transition` definition generally includes some `triggering events`. In our case, the `triggering events` may be given by the conditions under which the change of state values happens. As an example, the portion of code in Figure 4, corresponds to two possible state transitions described in Figure 7. A transition from a state $s_1$ to a state $s_2$ where `devxx_cs_unavailable` is set and a transition from the state $s_1$ to a state $s_3$ where `devxx_cs_idle` is set. The triggering event of the first transition is given by the condition `(party.status == _dialing && party.call_state != ringing)` while the second transition is triggered by the negation of this condition.

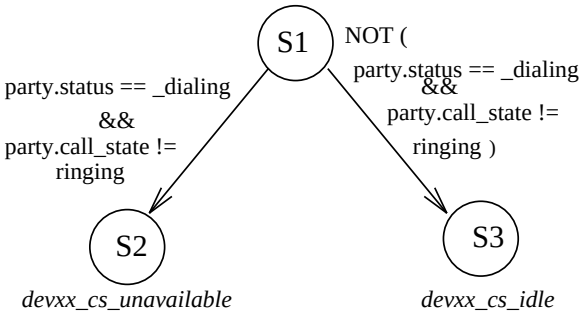


**Figure 7. State transitions corresponding to Figure 4.**

The complete finite state machine model of a call process may be recoverred by finding all its states and states transitions. The set `state(P)` gives all the `possible` call processing states of the process `P`. However only a subset of `state(P)` corresponds to real states which the process can enter.

We do not extract a complete state model in the PBX system. Given the size of this system, the state model would be extremely complex, the number of possible states and transitions in this system being huge. Therefore, we believe that the value of this state model in understanding the system would be undermined by its size and complexity.

We rather extract pieces of information about states that provide facets of call states. The state model obtained does not make a single global state diagram. We get several views of the system states useful for incremental browsing and understanding of the source code.

### 3.4. Extracting pieces of state information

We extract, pieces of state information to augment the database of TKSEE and then support state based exploration.

The extracted information describes :

- Where state variables are defined.

  This includes declaration as formal parameters and also declarations as global variables in call processes.

- How state variables are used.

  We record every use of state variables including :

  - setting to state values,
  - comparison with state values,
  - use for control through `if` and `case` statements,
  - etc.

- Where state variables are used.

  That includes knowing the use of state variables as actual parameters.

We find state variables by parsing the source code. But the approach supposes having a knowledge of the call state types. In the case of the PBX system used for our experiment, this knowledge is provided by a naming convention.

Tracking how state variables are used is more complex because state variables may be set or compared to other state variables. In such cases, the information in the data base specifies the fact that the state variable is set/compared to another state variable.

The state information extracted allows new source code exploration and understanding facilities within TKSEE. It becomes possible to combine queries about state variables with more “traditional queries”. As examples, a software maintainer can issue queries to ask :

- in which routines a given state variable is modified ?,
- what are the routines called when a state variable has a specific state value ?,
- what are the possible transitions for a given state variable ?,
- etc.

By issuing a series of queries, a maintainer can follow the evolution of a state variable through different stages of a process call graph and then acquire a knowledge about the effect of this variable. The overal result of this kind of exploration is similar to the inference of a state transition model according to a facet of the system given by this state variable. However this state model is contructed *mentally* by the software maintainer through the exploration.

### 3.5. Automatic derivation of state models related to single variables

State based exploration helps maintainers mentally make up state models.

It is however useful to automatically derive state models that show the transitions of single state variables. An interest of such state models is to help newcomers learn a system. State models related to single state variables will also be smaller and less complex than complete state models as discussed in section 3.3.

Automatic (non assisted) derivation of a state model supposes knowing the state values set to state variables in all the cases. Unfortunately, state variables are sometime set to other state variables, or even to values returned by function calls. It is possible to use an inter-procedural slicing technique [7, 14] to know state values set to state variables in

these situations. Program slicing [18, 1] is a technique to extract from a program the statements that affect a given computation. Slicing gives answers to questions such as "what statements potentially affect the computation of a variable *v* at statement *s* ?"

Suppose an assignment statement where a state variable $sv_1$ is set to the value of a state variable $sv_2$. Program slicing can help finding all the other computations which involve $sv_2$ and then determine what may be the value of $sv_2$ in the assignment. Slicing is however costly in time and space. Our current work includes making this technique effective for large software systems such as ours.

## 4. Conclusion

Programs in distributed telecommunication systems are based on concepts such as `state` and `state transition`. In this paper, we discussed an approach to get state information from source code, and enhance program exploration.

Our approach is based on a particular PBX system. The peculiarities of the approach linked to this system concern the assumptions that `state variables` are used to describe states and a special naming convention is used to find `state types` that define these state variables. We believe that our approach can be generalized because :

- The natural way to map a state model to code is to use state variables to model states and case/if-like statements for state transitions. Therefore, it is reasonable to believe that the approach used in our PBX system is common to lot of similar systems.

- In order to be manageable, any large state based system which uses state variables should use some convention to distinguish state variables (and/or state types) to other variables. A naming convention is used in our system, other conventions may be used in other systems.

  In any case, our approach essentially needs the set of all the state types. In absence of a convention, such information can be obtained from the experts of the system.

State is only one particularity of distributed telecommunication systems. Another aspect of these systems is inter process relationships. As an example, in our PBX system, call processes communicate intensively each with the other in the Main Controller by inter process message exchanges. There are also parent/child relationships between call processes with some call processes creating or destroying other call processes. These inter process relationships reflects both on the architecture and the behavior of the system. Our work includes abstracting inter process relationships from source code and developing exploration capabilities based on these.

We see state based program understanding discussed in this paper and program understanding based on inter processes relationships as complementary. State based program understanding helps in understanding processes taken in isolation while inter-process relationships provide a high level view of the whole system behavior by showing the collaboration of independent processes entities.

Finding inter process relationships by a static analysis of source code is however a difficult task [8, 6, 11]. The difficulty comes from the fact that inter process relationships are often established at runtime. We distinguish two degrees in inter-process relationship recovery : `static` and `dynamic` recovery. Static recovery concerns inter-process relationships apparent in the source code while dynamic recovery considers links between running processes. We are considering system run traces to help find dynamic inter-process relationships.

## References


[1] D. Binkley and K. Gallagher. Program slicing. *Advances of Computing*, 43, 1996.

[2] R. Büchi. On a decision method in restricted second-order arithmetic. In *Proc. International Congress on Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[3] Y. Choueka. Theories of automata on $\omega$-tapes: a simplified approach. *J. Comput. System Sci.*, 8, 1974.

[4] K. B. R. E. group. Introduction to tksee 2.0. http://www.csi-.uottawa.ca/ tcl/kbre/tksee.html.

[5] R. Holt. Software Bookshelf: Overview And Construction. http://www.turing.toronto.edu/ holt/ papers/bsbuild.html.

[6] L. J. Holtzblatt, R. L. Piazza, H. B. Reubenstein, S. N. Roberts, and D. R. Harris. Design recovery for distributed systems. *IEEE Transactions on Software Engineering*, 23(7):461–472, July 1997.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), Jan. 1990.

[8] T. Kunz and J. P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transactions on Software Engineering*, 21(6), June 1995.

[9] T. C. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. Technical Report TR-97-07, University of Ottawa, Computer Science, Dec. 1997.

[10] T. C. Lethbridge and J. Singer. Strategies for Studying Maintenance. In *Workshop on Empirical Studies of Maintenance*, pages 79–84, Monterey, California, Nov. 1996.

[11] N. C. Mendonça and J. Kramer. Developing an Approach for the Recovery of Distributed Software Architectures. In *IWPC'98, 6th International Workshop on Program Comprehension*, pages 28–36, Ischia, Italy, June 1998. IEEE Computer Society.

[12] H. Müller, M. Orgun, S. Tilley, and J. UHL. A Reverse-engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[13] Power Software Corporation home page. http://www.power-soft.co.uk/.

[14] T. Reps and G. Rosay. Precise Interprocedural Chopping. In *Proceedings of the 3rd ACM Symposium on Foundations of Software Engineering*, Washington, DC, Oct. 1995.

[15] S. Rugaber. Program Comprehension. *J. Encyclopedia of Computer Science and Technology*, 35, 1996.

[16] Take5 Corporation home page. http://www.takefive.com/index.htm.

[17] J. Weidl, R. Klösch, G. Trausmuth, and H. Gall. Facilitating program comprehension via generic components for state machines. In *5th Int. Workshop on Program Comprehension*, pages 118–127, May 1997.

[18] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, SE-10:352–357, July 1984.