

In order to establish a comparison point for the detection quality of the automatic recovery techniques, software engineers manually compiled a list of **reference components** (short *references*) for diverse C systems. These reference components are used for statistical analyses, for calibrating parameters of diverse metrics, and to evaluate the automatic techniques. For the evaluation, we compared the components proposed by automatic techniques, called **candidate components** (short *candidates*), to the reference components. The sets of references for this comparison are called the **reference sets** or **reference corpora**. This section summarizes how the reference sets were obtained and validated.

1 Systems Studied

The reference components were obtained for several medium size C programs (see Table 1 for their characteristics). *Aero* is an X-window-based simulator for rigid body systems (Keller, 1995), *Bash* is a Unix shell (Ramey, 1994), *CVS* is a tool for controlling concurrent software development (Berliner, 1990), and *Mosaic* is a world-wide web browser (NCSA, 1997).

Table 1. Suite of analyzed C systems.

System Name	Version	Lines of Code	# User Types	# Global Objects	# Routines
<i>Aero</i>	1.7	31 Kloc	57	480	488
<i>Bash</i>	1.14.4	38 Kloc	60	487	1002
<i>CVS</i>	1.8	30 Kloc	41	386	575
<i>Mosaic</i>	2.6 (without GUI)	37 Kloc	79	269	564

All figures about program length in terms of *lines of codes* throughout this thesis are ascertained with the Unix tool *wc*, hence include comments and blank lines. Most systems have additional libraries that often encapsulate platform dependencies. These libraries were not investigated. Table 1 lists only the size of the core systems that were analyzed by the software engineers.

2 Obtaining the Reference Components

The reference components of *Aero*, *Bash*, and *CVS* were compiled by human analysts in 1997. The reference components for *Mosaic* are the result of the experiment conducted by Koschke (2000). The actual numbers of all major forms of atomic components (abstract data types, abstract data objects, hybrid atomic components) that were identified for each studied system are listed in Table 2. The rest of this section gives more detail about how the reference components were established for the respective systems and why they provide a reasonable basis for comparison.

Table 2. Number of atomic components in analyzed systems.

System	#ADT	#ADO	#Hybrid	#Total
<i>Aero</i>	9	16	1	26
<i>Bash</i>	18	16	5	39
<i>CVS</i>	13	35	6	54
<i>Mosaic</i>	12	28	13	53

2.1 Reference Components for *Aero*, *Bash*, and *CVS*

We asked five software engineers to identify atomic components in *Aero*, *Bash*, and *CVS*. Table 3 summarizes their experience and how the task was divided among them.

Table 3. Human analysts.

Software Engineer	Programming Experience	System Analyzed
se1	2 years research	<i>Bash</i>
se2	2 years research	<i>Bash</i>
se3	5 years research	<i>Bash</i>
se4	5 years research	<i>CVS</i>
se5	> 5 years industry	<i>Aero</i>

There was no overlap of their work. They needed between 20 and 35 hours for each system to gather the atomic components of the respective systems. The software engineers were provided with the source code of each system, a summary of

connections between global variables, types, and functions, and the guidelines given in Figure 1.

Identify the existing atomic components present in this system. These are abstract data objects (ADO), and abstract data types (ADT), or a combination of both.

- An **abstract data type (ADT)** is an abstraction of a type which encapsulates all the type's valid operations and hides the details of the implementation of those operations by providing access to instances of such a type exclusively through a well defined set of operations
- An **abstract data object (ADO)** is a group of global variables and constants together with the routines which access them.
- In real programs, we often find mixtures of ADTs and ADOs, i.e., components that contain both types as well as variables. These components are called **hybrid components**.
- The key difference of ADT and ADO is that an ADT is built around a type and an ADO around a set of simple global variables. This can be decided automatically, so do not waste time writing it down. Just identify the **functions, variables, and types which belong together** because they are cohesive and correspond to the idea of **ADO and/or ADT**.
- In practice, programmers sometimes break the encapsulation principle, therefore we widen the definition of abstract data objects and abstract data types to clusters of types or variables, respectively, with their accessor routines. The internal representation of ADTs and ADOs can be public.
- Nota bene: not all functions, variables, or types have to be put into ADO, ADT, and hybrid components.
- In general, your experience and understanding has more value than rules, you are the last judge of what constitutes an ADO/ADT.

Figure 1. **Guidelines for human analysts.**

Because the analysis of *Bash* was distributed among three software engineers, they performed a review of each other's work and came to a consensus on the final reference components.

The guidelines did not exclude overlap between atomic components, i.e., sharing elements among components. There was a small degree of overlap of the reference components of *Aero* and *Bash* and no overlap for *CVS*.

The fact that our reference components used as comparison point were produced by people raises the question whether other software engineers would identify the same atomic components. In order to answer this question, Jean-François Girard performed an experiment on a subset of *CVS* containing 2.8 KLOC and composed of the following key files: *history.c*, *lock.c*, *cvs.h*. These source files were distributed along with a cross-reference table indicating the relations among types, global variables, and functions. Four software engineers had the task to identify the atomic components present. He collected a description of the procedure they followed along with their results, then looked for cases where they seemed to have broken their own rules and asked them to refine either their procedure or their results. He also revisited with them those atomic components for which a comment indicated that they were unsure or something was unclear and corrected their results according to their conclusions.

The four software engineers agreed on the basic principles that characterize an atomic component and proposed very similar components. There were some divergences on the details; for example, one of them added functions to an abstract data type which did not have the type *T* of the abstract data type in their signature, but applied a cast of type *T* to one of their parameters. These divergences occurred rarely.

Jean-François Girard (Fraunhofer institute for experimental software engineering in Kaiserslautern, Germany) performed a second experiment on a subset of *Bash* containing 5.9 KLOC and composed of the following key files: *copy_cmd.c*, *dispose_cmd.c*, *execute_cmd.c*, *make_cmd.c*, *print_cmd.c*, and *command.h*. He followed the same procedure but distributed the subsystem to two software engineers who did not know the system to avoid learning effects.

Finally, in order to assess if these experiment results from a system subset can be generalized to a complete system, one software engineer identified atomic components in the whole *Bash* system. The atomic components he identified were compared to those of the reference components used in this thesis (those obtained by consensus).

A quantitative evaluation of the degree of agreement among the software engineers showed first, that the software engineers agreed to a very high degree on the atomic components of these systems and second, that the agreement gained on a smaller subset can indeed be generalized to the rest of the system. Therefore, we may conclude that the reference components for *Aero*, *Bash*, and *CVS* are a suitable oracle. The procedure used for the quantitative evaluation and its exact results are explained by Girard, Koschke, and Schied (1999).

2.2 Reference Components for Mosaic

The reference components for Mosaic were obtained by an experiment conducted by Koschke (2000), in which human analysts had to detect atomic components either manually or with tool support. The task of the experimental subjects was to recover as many atomic components for *Mosaic* as possible within 6 hours. In order to obtain comparable results, we reduced the possible search space for components to a size that could be handled within the given time frame, i.e., all experimental subjects should be able to look at all source files within the available time. Therefore, we excluded the files of *Mosaic* that are mainly devoted to the graphical user interface, namely, all files whose names begin with the prefix *gui*. The 8 excluded files comprise 15 KLOC, i.e., 40 files consisting of 37 KLOC were to be analyzed. To obtain a common basis of comparison, the components separately detected by each individual were merged and then validated by at least two participants. Only those components were accepted for which a consensus could be reached.

Nine students volunteered for the experiment. Since we chose a two-block design for the experiment, we asked the system administrator of our department to participate in order to get equal group sizes. The system administrator was not involved in programming for our project.

At the time of the experiment, the students were studying computer science at the University of Stuttgart (six at the graduate level, three at the undergraduate level). All of them had at least two years of programming experience and were familiar with the programming language C. See Table 4 for their individual profile.

Table 4. Profile of the experimental subjects.

experimental subject	# semester	programming experience
S ₁	11	good
S ₂	9	good
S ₃	9	good
S ₄	3	good
S ₅	5	average
S ₆	9	good
S ₇	3	good
S ₈	3	average
S ₉	9	good
S ₁₀	professional	good

3 References

Berliner, B. (1990), ‘CVS II: Parallelizing Software Development’, Proceedings of 1990 Winter USENIX Conference, Washington, D.C.

Girard, J.F., Koschke, R., and Schied, G. (1999), ‘A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation’, Journal on Automated Software Engineering, no. 6, October, pp. 357-386, Kluwer Academic Publishers.

Keller, H., Stolz, H., Ziegler, A., and Bräunl, T. (1995), ‘Virtual Mechanics Simulation and Animation of Rigid Body Systems with Aero’, Simulation for Understanding, vol. 65, no. 1, pp. 74–79, July.

Koschke, R. (2000), 'Atomic Architectural Component Recovery for Program Understanding and Evolution', Ph.D. Thesis, Institute for Computer Science, University of Stuttgart.

NCSA (1997), 'NCSA Mosaic Home Page', National Center for Supercomputing Applications, <http://www.ncsa.uiuc.edu/SDG/Software/Mosaic>.

Ramey, C. (1994), 'Bash - The GNU shell', Linux Journal, issue 3, August, Specialized Systems Consultants.