Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering

Aoun Raza, Gunther Vogel, and Erhard Plödereder

Universität Stuttgart
Institut für Softwaretechnologie, Universitätsstraße 38
70569 Stuttgart, Germany
{raza,vogel,ploedere}@informatik.uni-stuttgart.de

Abstract. The maintenance and evolution of critical software with high requirements for reliability is an extremely demanding, time consuming and expensive task. Errors introduced by ad-hoc changes might have disastrous effects on the system and must be prevented under all circumstances, which requires the understanding of the details of source code and system design. This paper describes Bauhaus, a comprehensive tool suite that supports program understanding and reverse engineering on all layers of abstraction, from source code to architecture.

1 Introduction

This paper presents an overview of the program understanding and reverse engineering capabilities of Bauhaus [1], a research project at the universities of Stuttgart and Bremen. The importance of understanding program source code and of being able to reverse engineer its components derives both from the costly effort put into program maintenance and from the desire to create and preserve the reliability of the code base even under extensive change. The quality of software under maintenance is crucially dependent on the degree to which the maintainers recognise, observe, and occasionally modify the principles of the original system design.

Therefore, tools and techniques that support software understanding and reverse engineering have been developed by industry and academia as a vehicle to aid in the refurbishment and maintenance of software systems. Especially critical systems with high requirements for reliability benefit from the application of such tools and techniques. For example, it becomes possible to automatically prove the absence of typical programming errors, e.g., uninitialised variables, to raise the internal source code quality, e.g., by detecting dead code, and to improve the understanding of the software on all layers of abstraction from source code to architectural design.

The details of these techniques will be described later in this document. The rest of the document is organised as follows: section 2 provides the motivation and background of Bauhaus. Section 3 discusses the program representations used in Bauhaus. Section 4 and 5 describe the low- and high-level analyses implemented in Bauhaus. Trace analysis techniques are introduced in section 6.

Some other analyses are discussed in section 7. Section 8 describes the development process and summarises experiences with Ada as the main implementation language of Bauhaus. The paper ends with some conclusions in section 9.

2 Background

The project Bauhaus is motivated by the fact that programmer efforts are mostly (60% - 80%) devoted to maintain and evolve rather than to create systems [2]. Moreover, about half of the maintenance effort is spent on understanding the program and data [3], before actual changes are made. Therefore, helping maintainers to understand the legacy systems they have to maintain could greatly ease their job. A better understanding of the code and of its design will undoubtedly also contribute to the reliability of the changed software. An important step in assisting the maintainers is to provide them with a global overview comprising the main components of the system and their interrelations and with subsequent refinements of the main components. Therefore, our initial goal for Bauhaus was the development of means to semi-automatically derive and describe the software architecture, and of methods and tools to represent and analyse source code of legacy systems written in different languages. At present, Bauhaus is capable to analyse programs in Ada, C, C++, and Java. Bauhaus is implemented mainly in Ada and interfaces to software in C, C++, and an assortment of other languages. Figure 1 provides basic data about the composition of the 2005 Bauhaus system.

Language	Handwritten	Generated	Total
Ada95	589'000	291'000	880'000
C	106'000	0'000	106'000
C++	115'000	177'000	292'000
. . .	. . .	. . .	. . .
Total	843'000	469'000	1'312'000

Fig. 1. The Bauhaus project: number of non-commented lines of code, categorised by programming language

The primary challenges that the Bauhaus infrastructure addresses are the support for multiple source languages and the creation of a common framework, in which advanced compiler technologies for data- and control-flow analyses offer foundation support engineered to allow the analysis of multi-million lines of user code. User-oriented analyses with ultimate benefits to the maintainers of systems achieve their goals by building on the results of these basic analyses. In the realm of tools for program understanding, Bauhaus is one of very few toolsets that takes advantage of data- and control-flow analyses.

3 Program representations

3.1 Requirements for program representations

The particular program representation has an important impact on what analyses can be performed effectively and efficiently. Fine-grained analyses, e.g., of data- and control-flow, require more low-level information than coarse-grained analyses. Some of the Bauhaus tools utilise compiler techniques, which produce rich syntactic and semantic information, often referred to as the detailed, low-level representation of a program. Unlike compilers, all Bauhaus tools analyse and know the system as a whole. The Bauhaus analyses used for reverse- and re-engineering the architecture of a system, on the other hand, build on a much coarser, high-level program representation. To reduce the size of the information being operated upon by these analyses, the detailed information is first condensed into this coarser, more suitable high-level representation. An additional design goal for our program representations was to keep them independent from the source programming languages and, in particular, to allow for the analysis of mixed-language systems.

In Bauhaus two separate program representations exist, catering to the need of detailed low-level and coarser high-level analyses, respectively. The *InterMediate Language* (IML) representation contains information at the syntactical and semantical levels. Resource flow graphs (RFG) are used to represent information about global and architectural aspects of the analysed systems.

3.2 IML

The IML representation is defined by a hierarchy of classes. Each class undertakes to represent a certain construct from a language, as for instance a while loop. Instances of these classes model the respective occurrences in a program. Within the hierarchy, a specialisation takes place: child classes model the same semantic construct as their parent class, however in a more specialised pattern, e.g., the While_Loop class is a child of the more general Loop_Statement class. By enforcing certain rules on the generation of the sub-nodes of such an instance, a semantic equivalence is ensured, so that analyses not interested in the fact that the loop was indeed a While_Loop will function correctly when operating on the instance merely as a general Loop_Statement. This modelling strategy allows us in many cases to add a construct from a particular language as instance of a more general common notion present in many languages. E.g., the for-loops of Ada and C are represented by two distinct classes. Both share the same base class that models the common aspects of all loops. In this regard, IML is quite unique among the known Intermediate Languages [4].

Objects in IML possess attributes, which more specifically describe the represented constructs. Often such an attribute is a pointer, or a list, or a set of pointers to other objects. Thus, IML forms a general graph of IML objects and their relationships. IML is generated by compiler frontends that support C and C++. Frontends for Ada and Java are under development. Foundation support

for the persistence of IML automates the writing and reading of IML to and from a file. Prior to advanced analyses, the IML parts of a program are linked by a Bauhaus tool into a complete representation of the entire program to be analysed.

3.3 RFG

As described earlier, different abstraction levels are used in Bauhaus for the recognition of the architecture of a software system. While IML represents the system on a very concrete and detailed level, the abstraction levels for global understanding are modelled by means of the RFG. An RFG is a hierarchical graph, which consists of typed nodes and edges. Nodes represent architecturally relevant elements of the software system, e.g., routines, types, files and components. Relations between these elements are modelled with edges. The information stored in the RFG is structured in views. Each view represents a different aspect of the architecture, e.g., the call graph or the hierarchy of modules. Technically, a view is a subgraph of the RFG. The model of the RFG is fully dynamic and may be modified by the user, i.e., by inserting or deleting node/edge attributes and types. For visualising the different views of RFGs, we have implemented a Graphical Visualiser(Gravis) [5]. The Gravis tool facilitates high-level analysis of the system and provides rich functionality to produce new views by RFG analyses or to manipulate generated views.

For C and C++, an RFG containing all relevant objects and relationships for a program is automatically generated from IML, whereas for Ada and Java the RFG is generated from other intermediate representations or compiler supported interfaces, e.g., the Ada Semantic Interface Specification (ASIS) or Java classfiles.

This RFG is then subjected to and augmented by additional automated and interactive analyses.

4 Analyses based on IML

This section describes the Bauhaus analyses that can be performed on the IML representation to help maintain the reliability and quality of the operational code.

4.1 Base Analyses for Sequential Code

In Bauhaus we have implemented the fundamental semantic analyses of control- and data-flow as well as different known and adapted points-to analysis techniques. The results of all these analyses are conservative in the sense that they produce overestimations in the presence of situations that are known to be undecidable in the general case. Examples are unrealizable paths or convoluted patterns of points-to aliasing on the heap. A different class of conservative inaccuracy comes from analyses which generate less precise results in favour of

a better run time behaviour in space and time consumption. As we strive to analyse large systems, these engineering tradeoffs are highly relevant.

Over the years, several pointer analyses with different characteristics have been implemented and evaluated in the context of the Bauhaus project. The focus of the first set of analyses was the production of highly precise points-to information for the data-flow analyses. The experiences with an implementation of the algorithm by Wilson [6] showed that the accurate results come at a high price. The run time and memory requirements of this analysis are often prohibitive. This cost as well as pronounced variations in timing behaviour prevented its application to large programs; even for smaller programs the runtime performance was too unpredictable. To be able to analyse programs with more than 200.000 lines of code, we have implemented flow insensitive analyses as developed by Steensgaard, Das, or Andersen [7–9]. These analyses show a much better and acceptable run time behaviour, but are considerably less precise than the analysis by Wilson. We presently investigate how those imprecise results can still be improved and the balance optimised between the cost of tighter results and their benefit to subsequent analyses.

The control-flow analysis in Bauhaus computes traditional intraprocedural control-flow graphs for all routines of a program. Together with the call-relationships, the set of intraprocedural control-flow graphs form an interprocedural control-flow graph that can be traversed by interprocedural data-flow analyses. The control-flow analysis is based on basic blocks which are a sparse representation of control-flow graphs. Besides the information of the possible flow of control between the basic blocks, derived information like dominance- and control-dependency information is available and subsequently used in architectural analyses.

A thorny issue for the analysis of C++, Java, and Ada is the representation of exception handling in control-flow graphs. Our model is similar to the modelling that was shown by Sinha and Harrold in [10]. As exceptions generate complex control-flow graphs and implicit exceptions might be raised at any time during an execution (e.g., the virtual machine error in Java), we compromised and consider only explicitly thrown exceptions in our analyses.

Data-flow relations are represented by the SSA-form (Static Single Assignment form). The SSA generation of Bauhaus is derived from the algorithm proposed by Cytron in [11]. The algorithm operates on locators which are an abstraction of variables or of parts of variables. The analysis itself does not know about the specific characteristics of the locators. Through this, it is possible to have analyses with different precision by just changing the locators, e.g., from one locator for each variable to one locator for each part of a structured variable. Also, it is possible to incorporate arbitrary pointer analyses by generating locators for the specific memory elements of each analysis. The data-flow analysis is performed interprocedurally in two phases. The first phase determines side effects, the second phase performs a local SSA generation for each routine and incorporates the side effects. The generation is context-sensitive, i.e., it takes

multiple calling contexts for each routine into account. Each calling context might result in different data-flow patterns and side effects.

Manifold applications for the results of the data-flow analysis exist. Simple tests for error detection like finding uninitialised variables, or the location of redundant or unread assignments, are truly trivial algorithms once the SSA form is available. Escape analysis for C pointer arguments is a slightly more elaborate but still simple algorithm.

Similarly, the results are the basis for applications of slicing or tracing for program understanding, all the way to applications on the architecture level like the recovery of glue code that implements the interactions between two components, or the classification of components within the architecture based on their external data-flow behaviour.

4.2 Base Analyses for Parallel Programs

In parallel programs, different tasks often need to communicate with each other to achieve their assigned job. Different communication methods are available for these interactions, such as message passing, use of shared memory or encapsulated data objects. Furthermore, tasks may need to claim other system resources that cannot be shared with others. As multiple threads try to access shared resources, their access must be protected by some synchronisation mechanism. Otherwise, their interaction could lead to data inconsistencies, which can further lead to abnormal program behaviour. Two important classes of inter-process anomalies are race conditions and deadlocks. A race condition occurs when shared data is read and written by different processes without prior synchronisation, whereas deadlock is a situation where the program is permanently stalled waiting for some event such as the freeing of a needed resource. Both these classes of errors tend to be very difficult to detect or to recreate by test runs; they arise in real-life execution as an inexplicable, sudden, and not recreatable, sometimes disastrous malfunction of the system. For reliable systems it is literally a "must" to impose coding restrictions and to perform a static analysis of the code to ensure the absence of race conditions and deadlocks. Tools can help to discover the situations and can assist programmers in locating the culprit source code.

There has been considerable research on defining different static and dynamic analysis techniques and building tools for race detection [12–14]. Tools based on static approaches need good base analyses, i.e., points-to and alias analyses [12]. In Bauhaus different points-to, control- and data-flow analysis techniques are implemented as discussed in 4.1. To overcome the deficiencies in previously proposed solutions we are now exploiting the Bauhaus base analyses for the implementation of race detection and deadlock analysis tools. A tool LoRad for the detection of data races in parallel programs has been implemented and is in its testing phase. LoRad uses the Bauhaus control-flow and points-to analyses to detect competing concurrent read and write accesses of variables shared by multiple threads or tasks, but executed without proper mutual exclusion. While not always an error, such accesses at best may cause non-deterministic program

results, at worst are truly disastrous if multiple, functionally related variables are updated and read without proper synchronisation.

It is worth mentioning that none of the already implemented race detection approaches have included rich points-to information, which is surprising, as the prevalent OS-interfaces are invariably based on pointer semantics for their arguments. Additionally, we are also implementing a deadlock detection technique, which uses a data-flow analysis based technique to check for necessary conditions to enable dangerous cyclic waiting situations.

4.3 Dead Code Analysis

Many systems contain code that is never executed because the corresponding subprograms are never reached. Despite being not necessary, the so-called dead code complicates the analysis and evaluation of the software and should be eliminated.

Bauhaus provides tools for the automatic detection of dead code. Those tools test the reachability of routines in the call graph. For safe and precise approximations of the effects of indirect calls, our tools consider the results of the points-to analyses described in section 4.1.

5 Analyses based on RFGs

This section describes some Bauhaus analyses performed on the high-level RFG representation.

5.1 Component Recovery

The IEEE standard for recommended practice for architectural description of software-intensive systems [15] defines architecture as the fundamental organisation of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. Bauhaus focuses on the recovery of the structural architecture that consists of components and connectors. 12 automatic component recovery techniques were evaluated, none alone has a sufficient recovery coverage [16]. To overcome the limitations exhibited by automated component recovery techniques, a semi-automatic approach was developed and implemented that combines automatic techniques in an interactive framework. The effectiveness of the method was validated through a case study performed on xfig [5]: In the limited time of five hours an analysis team was able to gain a 50% coverage of the source code, which was sufficient to understand the full architecture of xfig. Later the team could validate the acquired knowledge by solving three typical maintenance tasks.

5.2 Reflexion Analysis

Reverse engineering large and complex software systems is often very time consuming. Reflexion models allow software engineers to begin with a hypothetical

high-level model of the software architecture, which is then compared with the actual architecture that is extracted from source code. We extended the reflexion model defined by [17] with means for hierarchical decomposition [18], i.e., now the entities of the hypothetical and the actual architecture may contain other entities. The reflexion starts with a coarse model of the architecture which is iteratively refined to rapidly gain deeper knowledge about the architecture. Most importantly, the reflexion analysis flags all differences of the two models with respect to dependencies among entities. Thus, unintentional violations of the hypothetical model can be easily recognised or the model adjusted, respectively. Two major case studies performed on non-trivial programs with 100 and 500 KLOC proved the flexibility and usefulness of this method in realistically large applications. These systems, maintained and evolved over a long time, contained many deviations from the hypothetical architecture which were detected by our tool. In many cases, the deviations are not flaws, but rather a result of a too idealistic model. Thus, our tools help to find reality and avoid surprises by unrecognised dependencies. An interesting side-result of the case studies was that the quality of the results depended heavily on the availibility of points-to information.

5.3 Feature Analysis

Features are the realisation of functional requirements of a system. In trying to understand a program and its architecture, maintainers often want to know where in the code base a set of features has been implemented. For Bauhaus, we have developed a new technique for feature location [19]. A set of scenarios (test cases) invokes the features of interest and a profiler records the routines called by each test case. The relations between test cases and features and between test cases and routines is then analysed by concept analysis [20]. The output of the analysis is a lattice that allows a classification of the routines with respect to their specificity for the implementation of a particular feature or groups thereof. This assessment is of significant value in judging the effects of changes or the ability to extract a component from a system for reuse. The maintainer can additionally augment the concept lattice with information learned from the static call graph in Bauhaus. The very nature of deriving the lattice from test case profiles implies that important cases might be missed, but are guaranteed to be present in the static call graph. Inversely, the static call graph may include calls that implement cross-cutting concerns and hence are not specific to a feature.

5.4 Protocol Analysis

Despite being important aspects of interface documentation, detailed descriptions of the valid order of operations on components are often missing. Without such a specified protocol, a programmer can only guess about the correct use of the component.

The component recovery implemented in Bauhaus is able to discover the exported interface of components which serves as a starting point for further

analysis. The protocol recovery described by Haak [21] can be applied to discover the actually used protocol of the component. It is based on information derived from dynamic and static trace analyses (see section 6). The retrieved traces are transformed into finite automata which are later used in a unification process to form the protocol of the component.

The protocol validation [21] automatically detects infringements of protocol, e.g., if a component was accessed before it was initialised. The validation is based on an existing protocol which is compared with the actual use.

6 Static and Dynamic Traces

6.1 Static Trace Extraction

Traces are records of a program's execution and consist of sequences of performed operations [22]. Static trace graphs cover all possible executions and are derived directly from IML. In Bauhaus those graphs have many applications and are important input to further analyses, e.g., protocol recovery and validation (see section 5.4) and the recovery of information about component interaction. Static trace graphs are extracted with respect to an object that may be located on stack or heap [22] and contain all operations that might affect the object, including accesses, modifications and subroutine calls. In general, static trace graphs are projections of the interprocedural control flow graph and must cover all possible dynamic traces. Again, a key factor for the precision of the analysis is the quality of the base analyses. In four case studies performed using the Bauhaus implementation we further investigated these effects and showed that the extraction of trace graphs is feasible for programs with more than 100kLOC [22].

6.2 Dynamic Trace Extraction

As dynamic traces generally depend upon input, test cases have to be prepared that require a certain component for the derivation. Then, the source or object code program has to be instrumented [23], and executed on the specific input. The advantage of this dynamic analysis is that it yields precisely what has been executed and not an approximation. The problem of aliasing, where one does not exactly know at compile time what gets indirectly accessed via an alias, does not occur for dynamic analysis. Moreover, infeasible paths, i.e., program paths for which a static analysis cannot decide that they can never be taken, are excluded by dynamic analysis, too. On the other hand, the dynamic analysis lacks from the fact that it yields results only for one given input or usage scenario. In order to find all possible dynamic traces of the component, the use cases have to cover every possible program behaviour. However, full coverage is generally impossible because there may be principally endless repetitions of operations.

The Bauhaus dynamic analyses use IML for code instrumentation and further enhancements. The resulting IML graph is used to obtain a semantically equivalent C code. After its compilation, the execution of the program additionally generates traces in RFG representation. The generated traces are used in component and protocol analysis.

7 Other Analyses

7.1 Clone Detection

A widely used way of re-use is to copy a piece of code to some other place and possibly modify it slightly there. When an error is discovered in one of the copies, all other copies ought to be corrected as well. However, there is no trace of where the clones reside. Consequently, the same error gets rediscovered, reanalysed and fixed in different ways in each clone. Code quality suffers and the cost of maintenance rises. Studies have claimed that 20% and more of a system's code is duplicated in this fashion. The Bauhaus clone detection [24] identifies code clones of three categories: *type-I* clones are truly identical; *type-II* clones are copies in which identifiers or literals are consistently changed; *type-III* clones are modified by insertions or deletions and thus hardest to detect.

The evaluation of existing clone detection tools by Bellon [24] determined that a detection based on an abstract syntax graph has a better precision than a token-based clone detection. Consequently, the Bauhaus clone detection operates on IML.

7.2 Metrics

Metrics are quantitative methods to assess the overall quality of the software system and provide objective numbers for present and future software development plans. The metrics implemented in Bauhaus operate on different levels of a software system, i.e., source code or architecture level, and are computed on IML and RFG, respectively:

- **Source code level:** lines of code, Halstead, maximal nesting, cyclomatic complexity
- **Architecture level:** number of methods, classes and units, coupling, cohesion, derived metrics e.g., number of methods per class, classes per unit

The calculated results can be used in many ways, for instance, to estimate software complexity, to detect code smells, or to provide parameters to maintenance effort models. Most importantly, they can be used to observe trends while the software evolves.

8 Bauhaus development and experiences with Ada

We chose Ada as the main implementation language of Bauhaus, because we knew that few other languages would allow us to evolve and maintain a very large system in such a controlled and guided manner. The platform-independence of Ada allowed us to configure Bauhaus very easily to run on a variety of different platforms like x86-Linux, Microsoft Windows, and Sun Solaris.

The long term development and maintenance of Bauhaus in the research context is done by the researchers of the Universities of Stuttgart and Bremen.

Since the beginning, various student projects were performed on the Bauhaus infrastructure to implement new software components [25]. While most students would have preferred Java or C++ in the beginning, in retrospective (and after having worked on Java or C++ projects), they appreciated the features of Ada and it often seems to have become their language of choice.

Since all collaborators of the Bauhaus project have different backgrounds and are more or less experienced programmers, the adherence to common rules and standards is crucial. Since the introduction of the GNAT Coding Style, all developers share the same conventions which increased the readability and hence the quality of source code tremendously.

9   Conclusion

Bauhaus provides a strong and generic base for low- and high-level program understanding using advanced code- and data-flow analyses, pointer analyses, side-effect analyses, program slicing, clone recognition, source code metrics, static tracing, query techniques, source code navigation and visualisation, object recovery, re-modularisation, and architecture recovery techniques. In the near future we plan to extend Bauhaus with more analyses and error finding techniques for parallel programs. We have plans to implement deadlock and race detection analysis for Ada and Java. Bauhaus is growing as a large scale research initiative. Besides the University of Stuttgart we now have another Bauhaus working group at Bremen University. We have introduced portions of Bauhaus as a commercial product to deal with growing industrial response. Very recently, a company was created, focused on Bauhaus as a product.

Looking into the future, it is interesting to note that many program properties that were reasonably easy to analyse in procedural languages, because they were statically decidable, were moved in more modern languages into the realm of undecidability. For example, polymorphism makes determination of the called routine much more imprecise. Similarly, the efficiently decidable aliasing among reference parameters has now been mapped onto the undecidable and very difficult reference-value based aliasing. In short, object-oriented languages have significantly increased the need to obtain accurate points-to information, where in the past simple static semantic knowledge still sufficed. It is reassuring to know that, within Bauhaus, all tools can easily query the results of the IML base analyses to obtain this information.

References

1. Eisenbarth, T., Koschke, R., Plödereder, E., Girard, J.F., Würthner, M.: Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. In: 1. Workshop Software-Reengineering, Bad Honnef, Germany (1999)
2. Nosek, J.T., Palvia, P.: Software Maintenance Management: Changes in the Last Decade. Journal of Software Maintenance **2** (1990) 157–174
3. Fjeldstadt, R.K., Hamlen, W.T.: Application Program Maintenance Study: Report to Our Respondents. In: Proc. of GUIDE 48, Philadelphia, PA (1983)

4. Rainer Koschke, J.F.G., Würthner, M.: An Intermediate Representation for Reverse Engineering Analyses. In: Working Conference on Reverse Engineering, Hawaii, USA, IEEE Computer Society Press (1998) 241–250
5. Czeranski, J., Eisenbarth, T., Kienle, H., Koschke, R., Simon, D.: Analyzing xfig Using the Bauhaus Tool. In: Working Conference on Reverse Engineering, Brisbane Australia, IEEE Computer Society Press (2000) 197–199
6. Wilson, R.P., Lam, M.S.: Efficient Context-Sensitive Pointer Analysis for C Programs. In: PLDI. (1995)
7. Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM Press (1996) 32–41
8. Das, M.: Unification-based Pointer Analysis with Directional Assignments. In: PLDI. (2000) 35–46
9. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
10. Sinha, S., Harrold, M.J.: Analysis and Testing of Programs with Exception Handling Constructs. IEEE Trans. Softw. Eng. **26** (2000) 849–871
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transaction on Programming Languages and Systems **13** (1991) 451–490
12. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles, New York, NY, USA, ACM Press (2003) 237–252
13. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In: SOSP '97: Proceedings of the 16th ACM Symposium on Operating Systems Principles, New York, NY, USA, ACM Press (1997) 27–37
14. Helmbold, D.P., McDowell, C.E.: A Taxonomy of Race Detection Algorithms. Technical report, University of California, Santa Cruz, CA, USA (1994)
15. IEEE Standards Board: IEEE Recommended Practice for Architectural Description of Software-intensive Systems—Std. 1471-2000 (2000)
16. Koschke, R.: Atomic Architectural Component Detection for Program Understanding and System Evolution. PhD thesis, University of Stuttgart (2000)
17. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software Reflexion Models: Bridging the Gap between Design and Implementation. IEEE Computer Society Transactions on Software Engineering **27** (2001) 364–380
18. Koschke, R., Simon, D.: Hierarchical Reflexion Models. In: Working Conference on Reverse Engineering, IEEE Computer Society Press (2003) 36–45
19. Eisenbarth, T., Koschke, R., Simon, D.: Locating Features in Source Code. IEEE Computer Society Transactions on Software Engineering **29** (2003)
20. Lindig, C., Snelting, G.: Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In: Proceedings of the 19th International Conference on Software Engineering, IEEE Computer Society Press (1997)
21. Haak, D.: Werkzeuggestützte Herleitung von Protokollen. Diplomarbeit (2004)
22. Eisenbarth, T., Koschke, R., Vogel, G.: Static Object Trace Extraction for Programs with Pointers. Journals of Systems and Software (2005)
23. Larus, J.R.: Efficient Program Tracing. Computer **26** (1993) 52–61
24. Bellon, S., Koschke, R.: Comparison and Evaluation of Clone Detection Tools. IEEE Computer Society Transactions on Software Engineering **21** (2004) 61–72
25. Vogel, G., Simon, D., Plödereder, E.: Teaching Software Engineering with Ada95. In: Proc. Reliable Software Technologies, Ada-Europe 2005, York, LNCS (2005)