Auditdraw: Generating Audits the FAST Way

Neeraj K. Gupta
Lalita Jategaonkar Jagadeesan
Eleftherios E. Koutsofios
David M. Weiss

January 1997

In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, January 1997.

Auditdraw: Generating Audits the FAST Way

(In Proceedings of the IEEE International Symposium on Requirements Engineering, January 1997)

Neeraj K. Gupta[1], Lalita Jategaonkar Jagadeesan[2], Eleftherios E. Koutsofios[3] and David M. Weiss[2]

Abstract

Through a research/development collaboration, we have applied the FAST domain engineering process to the audits software in Lucent Technologies' 5ESS telephone switching system. Our collaboration has developed an application-oriented language, toolset, and accompanying process for specifying the requirements and generating the code for the 5ESS audits software. We describe the FAST process, our language, and the expected benefits of this project.

Keywords

requirements specification, requirements elicitation, software reuse, domain analysis, domain engineering, telecommunications, application-oriented languages, application generators, process, software engineering, requirements engineering

1 Introduction

Industrial software engineers continually face the question of how to produce their software faster, at lower cost, with more features. One approach to answering this question is to organize software into families, identify the requirements for the family, parameterize the requirements for individual family members, and to reuse assets within and across families to generate family members rapidly [9]. We describe here a collaboration between researchers and developers to apply a process, called FAST, that embodies such an approach. Our target family was the set of programs that audit the database in Lucent Technologies' 5ESS®telephone switching system to ensure that the switch operates reliably. Each member of the family is known as an audit. The FAST process guided us to develop a set of reusable assets that enable us to generate the C code for audits that is currently developed manually.

In particular, our collaboration has developed the following:

- an application-oriented language, called Auditdraw, designed especially for specifying the requirements and generating the code for the 5ESS audits software,
- a set of tools to help audits developers use Auditdraw, and
- a new process for developing audits using Auditdraw.

We expect that Auditdraw and its toolset and process will significantly increase productivity and significantly decrease cost and interval in audits software development in the 5ESS switch; we note, however, that it is not yet in production use.

Section 2 of this paper gives a brief description of Lucent Technologies' 5ESS switch and its audits software, section 3 describes the FAST process, and section 4 describes how we applied it to the audits domain. Section 5 contains our conclusions.

2 Lucent Technologies' 5ESS Telephone Switching System and Audits

Lucent Technologies' 5ESS telephone switching system [8] provides highly reliable telecommunications services; one of the key factors in ensuring system integrity and stability is the presence of reliable data.

In particular, audits [4] are programs that help ensure fault-tolerance of relational databases. In the 5ESS telephone switching system, these programs perform run-time checks on the consistency of data entities, and perform the appropriate corrections on data when an inconsistency is detected. For example, in a 5ESS switch, the status of units connected to the switch is maintained in the switch's database. Some units are arranged in hierarchies, and units that are

[1]Independent Consultant, with Lucent Technologies 5ESS Software Development Organization, `ngupta@dt2k.com`

[2]Software Production Research Dept., Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566 (USA) `{lalita,weiss}@bell-labs.com`

[3]Network Services Research Center, AT&T Labs – Research, 600 Mountain Ave, Murray Hill, NJ 07974 (USA) `ek@research.att.com`

related to each other by the hierarchy must have consistent status, e.g., if a parent unit is out of service, then all of its children must either be out of service or in a transient state.

A group of software developers are responsible for the 5ESS audits software. The requirements for audits come from developers from other subsystems, and are written in English. These requirements are typically given after the data design for a new or modified feature has been completed, and they specify the data entities that need to be audited, how to access those data entities, and what consistency checks to perform between different data entities. Requirements capture is followed by the construction of an audit design, and this design is subsequently reviewed at a high level design meeting and then a low level design meeting. Based on this design, the audit developers program the audit in C; this program is then reviewed in a formal inspection. The errors found during the code inspection are corrected and then the testing process begins. The first phase consists of unit testing, where the audits developers test the individual C functions. The second phase consists of integration testing, where other developers test all of the C code comprising the new feature – including code from other subsystems – by executing scenarios based on the requirements. This process is depicted in Figure 1.

With the encouragement of both research and development management, we formed a collaboration between researchers in the Bell Labs Software Production Research Department and developers in the 5ESS audits group to improve the audits development process. The collaboration has been applying the FAST process to the audits domain.

3 The FAST Process

The FAST process assumes that most software development is redevelopment, and that software production can be organized around families of systems to avoid much of the rework typically involved in redevelopment.[1] The goal of FAST is to provide a systematic approach to analyzing potential families and to develop facilities for efficient production of family members. Key to the process is finding the requirements for the family and appropriate abstractions for representing them, creating a language for specifying the requirements of individual family members, and then translating specifications of family members into deliverable software. Put another way, FAST is a systematic process for family-oriented, abstraction, specification, and translation.

FAST has two subprocesses, as shown in Figure 2.

- Defining the requirements for the family and developing a set of reusable assets for producing family members. This subprocess is known as domain engineering. We call its early phases domain analysis.

- Using the assets to produce family members, primarily by generation. This subprocess is known as application engineering.

The two subprocesses are connected by feedback loops to guide the evolution of the family and its assets.

3.1 Defining the Family and Developing the Reusable Assets

Defining the family means identifying the requirements for potential family members, characterizing what they have in common, and how they differ. For example, every audit must check for the existence of data before it attempts to access the value of the data. Furthermore, such a check must be done within the same time segment as when the data are accessed. All members of the family of 5ESS audits must obey this rule. On the other hand, the particular data to be accessed, the way in which the check is made, and the way in which the data are accessed vary over reasonably well-defined sets, and there are certain combinations that are not included in the family.

Just as one may organize requirements for single systems in a variety of ways, one may organize requirements for families in the same variety of ways. As an example, one might use categories such as interfaces to devices, interfaces to external systems, and behavior. Within each category (or further sub-categories), one may describe what's true for all family members (commonalities), what varies among family members (variabilities), and what the range of variability is. Section 3.1.1 describes the process that we use for eliciting such requirements and the artifact that results.

Note that no matter whether one prefers object-oriented approaches, functional approaches, or other approaches, one must decide what the potential family members are. We believe that this is equivalent to predicting what kinds of requirements changes are likely to occur during the lifetime of the family, and is of crucial importance in developing family members.

[1] FAST is a variant of Synthesis, which is described in [2, 10, 11]. The primary differences are that FAST does not have a separate activity for bounding a domain, and uses a much more structured approach for defining a family than does Synthesis. In addition, FAST relies more heavily on compiler-building technology than does Synthesis.

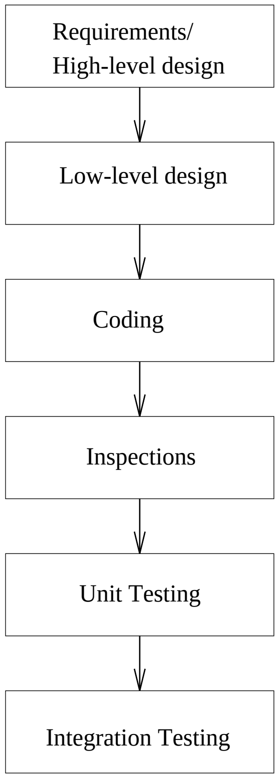

Figure 1: The Current Audit Development Process

3.1.1 Eliciting Requirements: The Commonality Analysis

We use a process called commonality analysis to elicit the requirements for a family. A commonality analysis is a structured, moderated discussion among a group of domain experts. Its result is a document, also called a commonality analysis [1], whose key parts include

1. A dictionary of terms used in discussing the family.

2. A list of assumptions that are true for all family members; these assumptions are known as commonalities and are requirements that every member of the family must meet. An example of a commonality is the requirement that every audit must check for the existence of data before it attempts to access the value of the data and that the check must be done within the same time segment as the access.

3. A list of assumptions about what can vary among family members; these assumptions are known as variabilities and are requirements that distinguish among family members. The way in which data are accessed by an audit is a variability.

4. A list of parameters that define, for each variability, the possible set of values that it can have and the time at which a value must be fixed when specifying a family member. The possible data access methods used by audits form the set of values for the variability described in the preceding paragraph. For a given audit, the access methods it uses must be declared when the audit is specified.

5. A list of issues that arose during the course of the analysis and, for each issue, a brief discussion of its resolution.

As much as possible, we use standard forms for expressing the terms, commonalities, variabilities, and parameters of variation. Except for the parameters of variation the standard forms are just structured prose. One example is commonalities that take the form “There is a mechanism for ...” An instance might be: “There is a fixed set of mechanisms that an audit may use for accessing data.” For parameters of variation we use a table that includes, as appropriate, mathematical descriptions of the value spaces of variabilities. Where standard forms do not fit, we use free

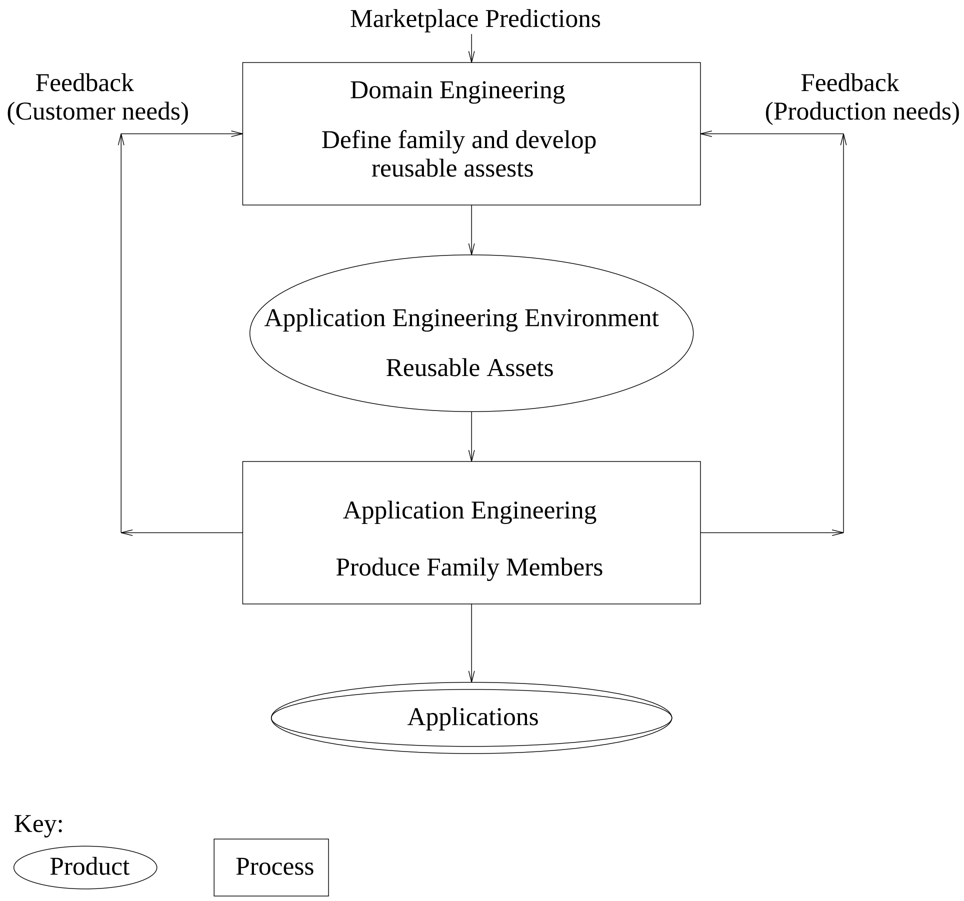

Figure 2: The FAST Process

text for describing terms, commonalities, and variabilities.

A commonality analysis document for a family provides the basis for designing a specification language and other reusable assets for the family.

The commonality analysis process is organized into phases that are designed to elicit terms, commonalities, variabilities, and parameters of variation by consensus from a group of 5-10 domain experts. Early phases of the process concentrate on gaining agreement among the domain experts on the objectives of the analysis and on the boundaries of the discussion. The intermediate stages of the analysis focus on gaining consensus for the definitions of commonly-used, important terms, for commonalities, and for variabilities. Later stages focus on parameterizing the variabilities and on reviewing the results of the analysis for completeness, consistency, and readability.

All stages of the commonality analysis are guided by a moderator who understands the FAST process, the role of commonality analysis within the FAST process, and the development culture in which the process is being used. Moderators have considerable discretion in adapting the process to different groups, but rarely change the structure of the artifact, i.e., we are flexible about the structure of the process, but inflexible about the structure of the document. A more detailed description of the commonality analysis process and artifact can be found in [12].

3.1.2 Reusable Assets

The reusable assets for a family consist of all the procedures, tools, and artifacts needed to produce family members, known in FAST as an application engineering environment. For example, a language for specifying family members and a translator for generating C code from a specification in the language are typically included in the environment. Those who use the environment follow a process specified by its developers. For 5ESS audits, the Auditdraw language and its translator form the initial environment. As the family of audits evolves, the environment will also, as the translator is enhanced and new tools are added to the environment.

3.2 Generating Family Members

The application engineering environment is designed to help its users to generate members of the family very rapidly. Much of its effectiveness depends on how accurately requirements for potential family members were predicted during domain analysis. When predictions about what family members will be needed in the future are accurate, the environment will be very effective. For this reason, a key input to the family definition process is predictions about marketplace trends.

Key to the environment is a well-designed language for specifying requirements. Its users should be able to specify particular family members just by specifying the variations considered during the definition of the family. For example, they should be able to specify for audits the data to be accessed by an audit. The language should allow them to do so in a way natural to the family, i.e., using the abstractions, such as data item fetch, that are used to define the family. The environment should provide them with facilities for verifying the choices they have made, e.g., verifying that all values for a particular data item have been checked.

The environment embodies both the process for creating family members envisioned during the definition of the family and the tools, procedures, and artifacts needed to carry out that process. Its users create a model of the family member that they would like to produce and then generate the family member. For 5ESS audits, the model is a specification expressed in Auditdraw. Generation of the family member is accomplished by supplying the specification to the Auditdraw translator, which performs completeness and consistency checks and generates the appropriate code.

3.3 Applicability of FAST

The FAST process is worth applying when the cost of domain engineering is repaid by the decrease in cost and development time for future family members, i.e., when the domain engineering cost can be amortized over the family members that are produced with the results of domain engineering. Such repayment occurs in the following situations:

- When a system will exist in many variations over a long period of time,
- When there is considerable time and effort being devoted to making continual changes to a system,
- When there are many customers for a system, each of whom wants the system customized for his or her purposes,
- When it is important to produce variations on a system quickly.

Much of our experience in applying FAST has been in legacy systems that are still in demand, where there is a reservoir of knowledge about the system, and where change to the system has become slow and costly compared with marketplace demands. Users of FAST often view it as a way to gain a competitive advantage in speed and cost.

We usually apply FAST by seeking a domain within a large, legacy system where there is frequent change occurring at relatively large cost. Such a domain is often an isolatable section of the system where the changes can be encapsulated, and where a group of software developers has responsibility for making the changes. Section 4 describes the application of FAST to such a domain within the 5ESS software. Although this is a typical application of FAST, we also believe it will work wherever developers are able to make informed decisions about family requirements.

3.4 Organizing FAST Applications

In addition to performing a commonality analysis, the FAST domain engineering process includes activities for designing and implementing the application engineering environment, and the application engineering process for using the environment to produce applications. A detailed description of these activities is beyond the scope of this paper.

As shown in Figure 2., we perform domain engineering and application engineering iteratively, reanalyzing, refining, and improving the environment as necessary. For the early iterations, we generally establish a collaboration of researchers and software developers to develop the initial version(s) of the environment. For the commonality analysis, the moderator is frequently a researcher teamed with 5-10 domain experts. For language design and implementation activities, the team is often composed of one researcher and two or three developers.

As prototypes of the environment become available, more developers are added to the team as testers. As the environment becomes ready for production, a wider set of developers is trained in its use, and researchers take a decreasing role in further environment development. We expect that the domain experts will become the owners of the environment and continue its development based on the feedback they get from its use and from forecasts of marketplace needs. Most of the domains that we are currently engineering are still in their first iteration of domain engineering.

Our experience, which is still very limited, indicates that the resources needed to develop the first version of an application engineering environment suitable for production use is less than five staff years of effort. We consider these domains to be in the first major iteration of the domain engineering-application engineering cycle shown in Figure 2. We currently have approximately ten domains somewhere in their first major iteration.

4 FAST and Audits

4.1 Domain Analysis and Application-Oriented Language Development

The audits software development group and several researchers collaborated to perform a domain analysis for the 5ESS audits domain, the first step in the FAST process. Thus, the requirements described in the commonality analysis [1] have been validated by a large group of audits experts.

For the language development phase, a member of the 5ESS audits software development organization visited the Software Production Research Department for approximately one year and a half. This interaction enabled a very fruitful and crucial exchange of ideas and concerns between research and development. Prior to our collaboration, there had been a prototypical rule-based language developed in the audits software group, on which we have capitalized.

Based on the domain analysis and this previous language prototype, we developed the Auditdraw language [3], designed especially for specifying the requirements for the 5ESS audits software. In the course of our work, we discovered that audit requirements can be very naturally represented as a form of decision trees, in which a decision involves the retrieval of a data entity from a database, and the comparison of its value to the value of some other data. Since some sub-trees may be identical, we have generalized these decision trees to directed acyclic graphs. The leaves of these graphs represent reports to be generated and corrective actions to be taken. Since these graphs correspond to audit requirements, the decisions, reports, and actions all have precisely specified behavior.

Auditdraw is a language for specifying such decision graphs. In the graphical view, the programmer interactively draws the graph on either a workstation or a PC; he/she uses a graphical interface designed especially for audit graphs, which is built on the graphical layout tool "dotty" [5, 6]. In addition to the graphical specification, the programmer also writes a simple companion declaration section that specifies the name of the audit and its interfaces, the data to be audited, and the data access methods to be used. The graph and declaration section are then together automatically translated into semantically-equivalent executable code.

A simple Auditdraw graphical specification is given in Figure 3. This specification gives an Auditdraw representation of a small piece of an audit in the 5ESS telephone switching system, which checks the consis-

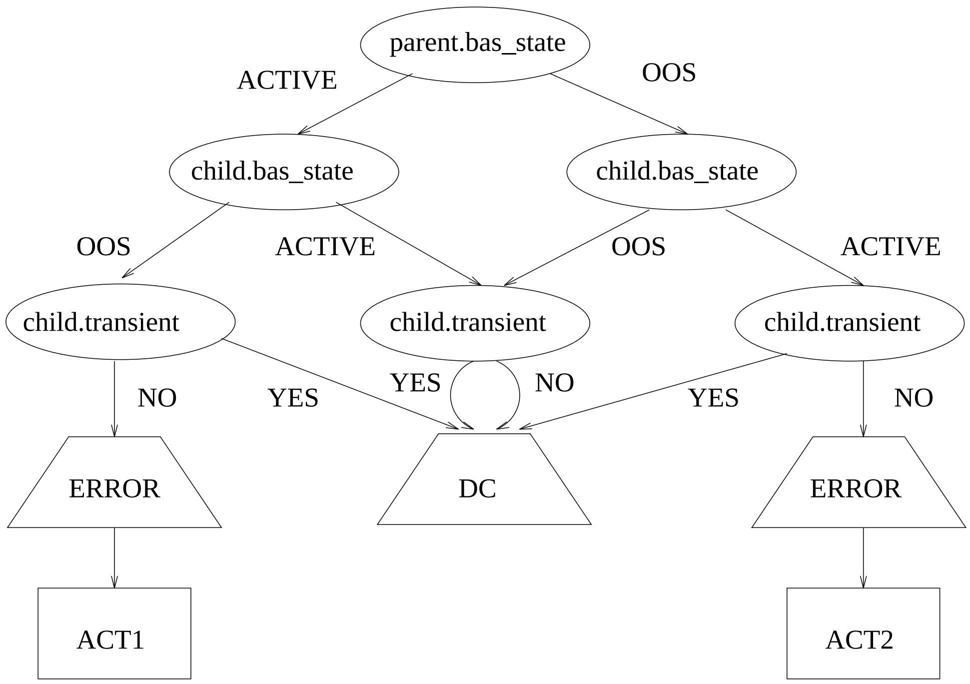

Figure 3: An Auditdraw Example

tency of parent and child circuits appearing in hierarchical circuits. Specifically, the states of a parent circuit and its child circuit should be identical: they should either both be active (ACTIVE) or both be out of service (OOS). If not, then an ERROR report should be generated and some corrective action (ACT1 or ACT2) taken. The only exception is when the child circuit is in a transient state, in which case any inconsistencies should be ignored and a "don't care" (DC) report should be generated. In our sample specification, the ovals represent the data entities to be checked (and the application of their associated retrieval method) and the arcs represent transitions. An arc is traversed if the value of the data entity in the source oval of the arc matches the value written on the arc; control is then passed to the target oval of the arc. Trapezoids represent error reports to be generated and boxes represent actions to be taken. So, for example, if the parent state is ACTIVE, the child state is OOS and the child is not in a transient state, then the ERROR report will be generated and the action ACT1 performed.

As illustrated above, Auditdraw specifications have a precisely defined semantics that model the behavior of audits. This enables static analysis – such as completeness checking and optimization – to be performed. For example, Figure 4 depicts an optimization of the Auditdraw graph shown in Figure 3; the first check performed in the optimized version is whether or not the child circuit is currently in a transient state. If so, the graph is exited immediately after generating a DC report.

Two views are supported by Auditdraw: a graphical view, described above, and a rule-based view. In the rule-based view, the programmer explicitly writes every maximal path of the graph in a textual format. These two views are interchangeable: a specification written in the rule-based view can be automatically translated into the graphical view, and vice-versa. Both of these views can be automatically translated into semantically-equivalent executable code.

4.2 Current Status and Future Plans

The commonality analysis for the Audits domain [1] was completed in May, 1994. A majority of software developers from the 5ESS Audits software development group and several members of the Software Production Research Department participated in this analysis.

The design of (both views of) the Auditdraw language, and the development of the graphical toolset and code generator was completed in July, 1995. This toolset is currently under trial in the Audits development group. As part of this trial, we have developed a Auditdraw graph specifying an actual 5ESS software audit; this graph consists of 56 nodes and 113 edges.

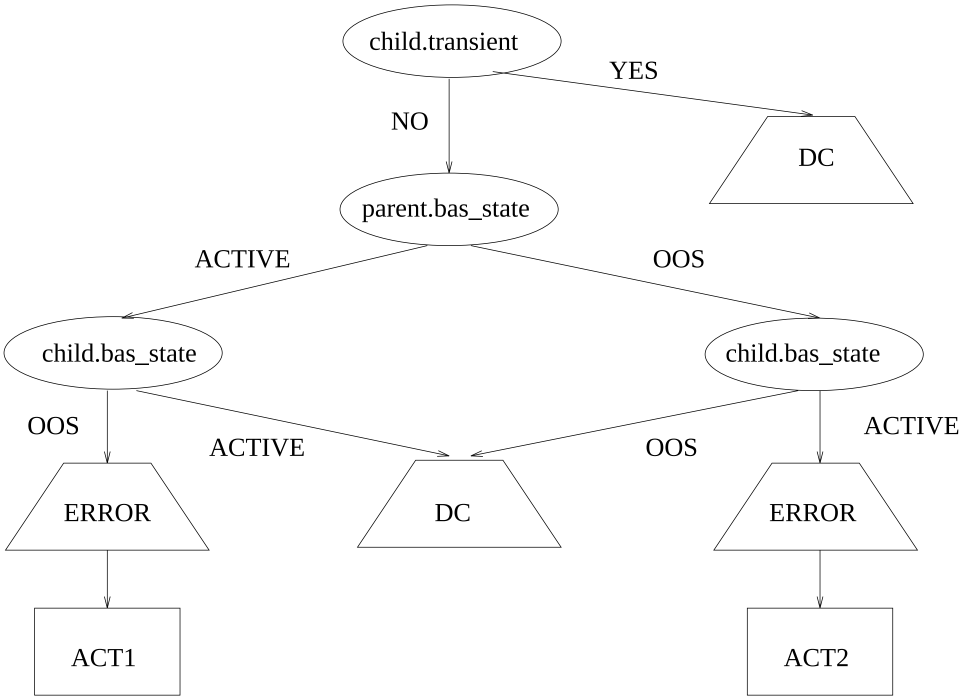

Figure 4: An Auditdraw Optimization Example

The C code automatically generated from this graph using Auditdraw consists of approximately 600 lines.

Since a formally-defined language and toolset now exist for the graphical view of Auditdraw, we plan to build on and extend this toolset after the completion of the trial. In particular, we plan to develop a toolset for the rule-based version of Auditdraw, as well as a debugger, optimizer, and "Auditdiff" tool for comparing the graphs/text of different audits specified using Auditdraw; this latter tool should be especially useful in the maintenance of audits. We also plan to develop a training course for Auditdraw.

4.3 Expected Benefits

The 5ESS audits software development group has begun a trial of the Auditdraw language and toolset. In this trial, several 5ESS audits will be specified using Auditdraw, and the resulting executable code will be tested in the 5ESS production environments.

Since Auditdraw is a high-level language designed especially for specifying audits, we believe that audits written in Auditdraw will be faster and easier to write and maintain. In particular, audit requirements can be specified directly in Auditdraw, and executable code can be automatically generated. We expect that this will significantly reduce coding errors, and that the resulting audits will be of a higher quality. Furthermore, automatic code generation from high-level specifications eliminates the need for several phases in the current development process: namely, the low-level design, coding, code inspection and unit testing phases. The Auditdraw process is shown in Figure 5; the reader should contrast this with the current audit development process shown in Figure 1.

As a side benefit of this streamlined process, we expect that audits will be available to other subsystems earlier, aiding in the debugging of those subsystems.

Thus, we believe that the Auditdraw language and toolset will significantly increase productivity and significantly decrease cost and interval in audits software development in the 5ESS switch. While Auditdraw is specific to the audits subsystem in 5ESS, it is also readily adaptable to other platforms. For example, the generated code could be re-targeted to a different language. More generally, Auditdraw is generic to databases; namely, it can provide fault-tolerance for a very large variety of data. We are currently investigating possible applications to other Lucent Technologies switching platforms.

5 Conclusions

Our application of FAST to the audits domain should significantly improve the efficiency of the audits development process. We believe this is a result primarily of the following factors:

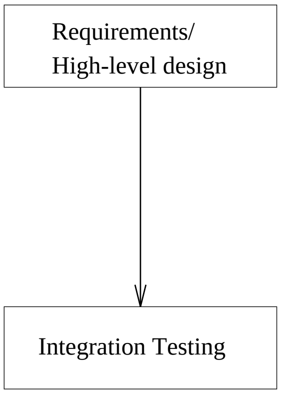

Figure 5: The Auditdraw Process

- *Family-oriented viewpoint*

  The FAST process is based on identifying the common requirements for a family of systems, parameterizing the requirements for individual family members, and finding the appropriate abstractions for easily expressing the commonalities and variabilities in requirements among family members. The commonality analysis process helps to ensure that these requirements and the abstractions remain correct and suitably expressive as the family of systems evolves.

- *Generation of code from requirements specifications*

  Requirements expressed in these abstractions are automatically translated into executable code. This streamlines the development process and ensures that the software satisfies the requirements.

- *Maintainability of software*

  Since the abstractions for expressing requirements are tailored for a particular family, in this case the 5ESS audits software, changes in the requirements are easily expressed by modifications in the requirements specifications. Furthermore, maintainability of the software in the face of changes in requirements is greatly aided by the automatic generation of code.

- *Static analysis of requirements specifications*

  The formally-defined nature of the graphical representation makes it possible to analyze statically the audits requirements for consistency and completeness, relative to the data that is being audited.

- *Reuse*

  Artifacts of the requirements engineering process can be reused in the following ways:

  Abstractions hold over large subsets of audits family members. In particular, commonalities are re-used in all the members of the family. For example, every 5ESS audit must check for the existence of data before it attempts to access the value of the data. Similarly, variabilities typically hold across large subsets of family members: for example, many audits share the same data access operations.

  The language translator reuses these abstractions for many audit designs, since the commonalities and variabilities are built into the language. For example, existence checks and appropriate data access operations are automatically inserted into audit designs during code generation.

  Abstractions are often shared between families. For example, we fully expect that many abstractions common to 5ESS audits will also be shared by audits in other switching platforms. Thus, Auditdraw and its process can be tailored to a variety of fault-tolerant databases.

In addition, we believe that the collaborative nature of the FAST projects, i.e., researchers and software developers working together to use the FAST process to create reusable assets, will greatly improve the chances for success. Not only did each type of collaborator bring specialized knowledge needed for the domain analysis and domain implementation, but we expect that transfer of the new technology that resulted from the collaboration will be greatly eased by having the technology users be part of the process.

Acknowledgments

We thank the 5ESS Audits group for many helpful discussions on this project, and Chris Ramming and Curt Tuckey for many helpful discussions on the language and toolset. We are grateful to Bob Colby, Cy Rubald, and Mary Zajac for their vigorous support of this collaboration, and Joe Paule and Eric Sumner, Jr. for their vision in initiating it.

References

[1] G. Babu, M. Baron, A. Charles, J. D'Mello, D. Ebright, N. Gupta, L. Jagadeesan, S. Ozdemir, S. Patel, J. Paule, M. Phreykz, R. Trygar, and D. Weiss. Commonality analysis for audits. Technical report, AT&T Bell Laboratories, May 1994. Internal Document.

[2] G.H. Jr. Campbell, S.R. Faulk, and D.M. Weiss. Introduction to synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium, 1990.

[3] N.K. Gupta, L.J. Jagadeesan, E.E. Koutsofios, and D.M. Weiss. User's guide for auditdraw. Technical report, AT&T Bell Laboratories, May 1995.

[4] G. Haugk, F.M. Lax, R.D. Royer, and J.R. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.

[5] E.E. Koutsofios and S.C. North. Viewing graphs with Dotty. Technical Report 59113-930120-04TM, AT&T Bell Laboratories, 1993.

[6] E.E. Koutsofios and S.C. North. Applications of graph visualization. In *Graphics Interface '94, Banff, Alberta*, pages 235–245, 1994.

[7] R.C.T. Lai and D.M. Weiss. A formal model of the FAST process. Technical Report Bell Labs Technical Memorandum, BL0112650-950707-30TM, AT&T Bell Laboratories, July 1995.

[8] K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July-August 1985.

[9] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2:1–9, March 1976.

[10] Synthesis guidebook, volume i, methodology definition. Technical Report SPC-91122-MC, v. 01.00.02, Software Productivity Consortium, December 1991.

[11] Synthesis guidebook, volume ii, case studies. Technical Report SPC-91122-MC, v. 01.00.02, Software Productivity Consortium, December 1991.

[12] D.M. Weiss. Defining families: The commonality analysis. Submitted to IEEE Transactions on Software Engineering, July 1996.