IMPROVING THE FRONT END OF THE SOFTWARE-DEVELOPMENT PROCESS FOR LARGE-SCALE SYSTEMS

G. David Bergland, Geoffrey H. Krader, D. Paul Smith, and Paul M. Zislis

For large-scale systems, diverse customer needs must be communicated and translated into precise implementations that are consistent with the existing system. Typically, we have relied on English-language text as a communications medium. But requirements written in English are often imprecise, and translating them into a finished product is largely a manual effort. As a result, the front end of the software development process—from analysis of customer requirements to high-level design—is generally time consuming, labor intensive, and error prone. In this paper, we discuss ways to apply computer-aided software-engineering technologies to the front-end process. We discuss ways to capture customer requirements in a machine-processable form where they can be reviewed, revised, analyzed, and refined into clear, unambiguous descriptions of what needs to be done and how to do it. Our goal in applying new and existing technologies is to mechanize much of the front-end process and thus reduce development costs and improve product quality.

G. David Bergland *is head of the Digital Systems Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey.* ***Geoffrey H. Krader*** *is head of the 5ESS® Switch Software Development System Department,* ***D. Paul Smith*** *is head of the 5ESS Switch Quality/Current Engineering Department, and* ***Paul M. Zislis*** *is head of the Advanced Software Technology Department. They are with AT&T Bell Laboratories (Indian Hill Park), Naperville, Illinois.*

Mr. Bergland is involved in research in software engineering, specification languages, programming environments, and photonic switching systems. He joined the company in 1966 and has a B.S., M.S., and Ph.D. in electrical engineering from Iowa State University.

Mr. Krader is responsible for the software development system for the 5ESS switch. He joined the company in 1975 and has a B.S.
(continued on page 21)

Introduction

The development of software for large, complex, real-time systems is a unique challenge. Both communications among people and communications among software modules must be carefully managed to meet customer expectations of quality.

But well-established methods (such as inspections and verification) to manage the quality of the product and well-established methods (such as written documentation) to manage communications are no longer enough to meet the challenge. Customers are demanding higher levels of performance, lower costs, and shorter development intervals. As a result, industry leaders are introducing new

Panel 1. Acronyms and Terms

CASE	computer-aided software engineering
CCITT	International Telegraph and Telephone Consultative Committee
FSM	finite-state machine
Inscape	specifies and checks interfaces between a system's functions as the system is being designed; uses formal interface specifications as input
ISDN	Integrated Services Digital Network
Kaleidoscope	uses axioms from formal language theory to deal with design objectives of event-driven systems and then mechanize their integration into a single system
POTS	plain old telephone service
SDL	Specification and Description Language; an international notation standard
SDL/GR	graphical, flowchart-like version of SDL
SDL/PR	programming-style text version of SDL
SVS	AT&T software for visual specification; provides graphical recording and animation of specifications
Statechart	graphical notation for finite-state machines based primarily on the idea of hierarchical sets of states
TELI	transportable English-language interpreter; a user-customized, natural-language processor
Watson	computes formal behavior specifications for process-control software—specifically telephone-call-control software—from informal "scenarios" that represent traces of typical system operation.

technologies—often labeled computer-aided software engineering, or CASE—to build in quality up front and assure that customer needs are met. (Panel 1 defines acronyms and terms used in this paper.)

In this paper, we discuss ways to apply CASE technologies to the front end of the software development process (i.e., from analysis of customer requirements through high-level design). We discuss the front end because it is much more cost effective to identify errors during this part of the development cycle than during later phases.[1] Among the technologies we discuss are formal specification techniques, artificial intelligence, visual programming, and animation. We also discuss management and organizational issues for the successful introduction of these new technologies.

In particular, we will focus on the application of CASE technologies to the development of AT&T's flagship 5ESS® switch. Of course, many of the challenges faced by the 5ESS switch's development team and many of their responses to these challenges apply to other large projects, as well.

Understanding Today's Development Process

The 5ESS switch is a large development effort—beyond the reach of many existing CASE technologies. The software that controls the switch contains several million lines of code. Incremental software releases can add several hundred thousand lines of code and require hundreds of staff-years of effort to develop.

Moreover, the 5ESS switch serves a diverse customer base. In the United States, it is used by each of the seven Regional Bell Operating Companies and several of the independent telephone companies. The switch is also serving customers in an increasing number of countries around the world. Each of these telephone administrations, whether in the U.S. or abroad, may have different requirements. If we fail to understand these requirements early in the development process, costly rework may be needed during later phases.

What we have here is a *communications* issue. Customer needs must be communicated to account managers, systems engineers, developers, and testers. Each of these groups of people has a different back-

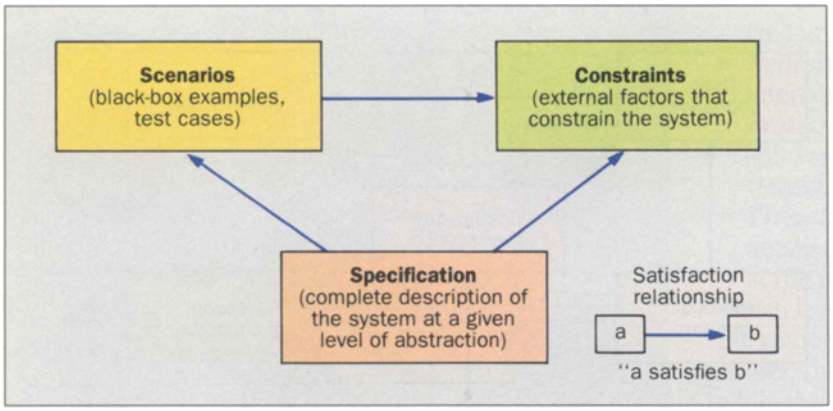


Figure 1. This formal model presents multiple views (scenarios, constraints, and specifications) of the software. Satisfaction relationships are always maintained; i.e., scenarios must satisfy constraints, and specifications must satisfy the scenarios and the constraints. These relationships support consistency checking and analysis.

ground, a different role in getting out the product, a different means of expression, and a unique perspective. All this complicates the communications process. So does the size of the project; the development team for a project as large as the 5ESS switch has an almost unmanageable number of communications paths.

Why Mechanize Information Capture? Traditionally, we have relied on natural-language (e.g., English) documentation as a communications medium. However, when written in English, requirements are often ambiguous and open to misinterpretation. This is an especially serious problem in developing telecommunications software, because the services provided often have complex interactions with other services. Other useful information, such as design intent, is often omitted altogether from the documentation. Moreover, the process of translating requirements into a finished product becomes time consuming and labor intensive, because natural-language documentation does not lend itself to mechanized analysis or consistency checking.

What is needed to improve this process is a way to capture information in machine-processable form early in the development cycle. Once captured, the information can be reviewed, revised, analyzed, and refined with relative ease. If the information is captured carefully, we can generate English-text documentation and test cases automatically from this information. As a result, documentation, source code (which must satisfy the test cases), and the original specifications could be synchronized and kept up to date.

Formal methods appear to be the key to capturing information, as does a set of human-friendly, cost-effective tools to support the formal methods. The rest of this paper discusses how formal methods can be applied and how CASE technology can be used—and is already being used—to support these methods.

Formal Models of Telecommunications Software

A formal model for representing the software is the focal point for improving communications during the front end of the software development process. Experience with software developers tells us that a good model must satisfy several requirements, including:

- It must support multiple views of the software.
- It must support multiple levels of abstraction.
- It must be machine processable.

Figure 1 depicts one such model, which is being tested on 5ESS switch software. The model provides three views of the software:

- *Scenarios*—a set of "black-box" examples or test cases of the software. A scenario might be a *script* for a specific call type (e.g., speed calling).
- *Constraints*—a set of external requirements on the software. A constraint might indicate the *style* of the

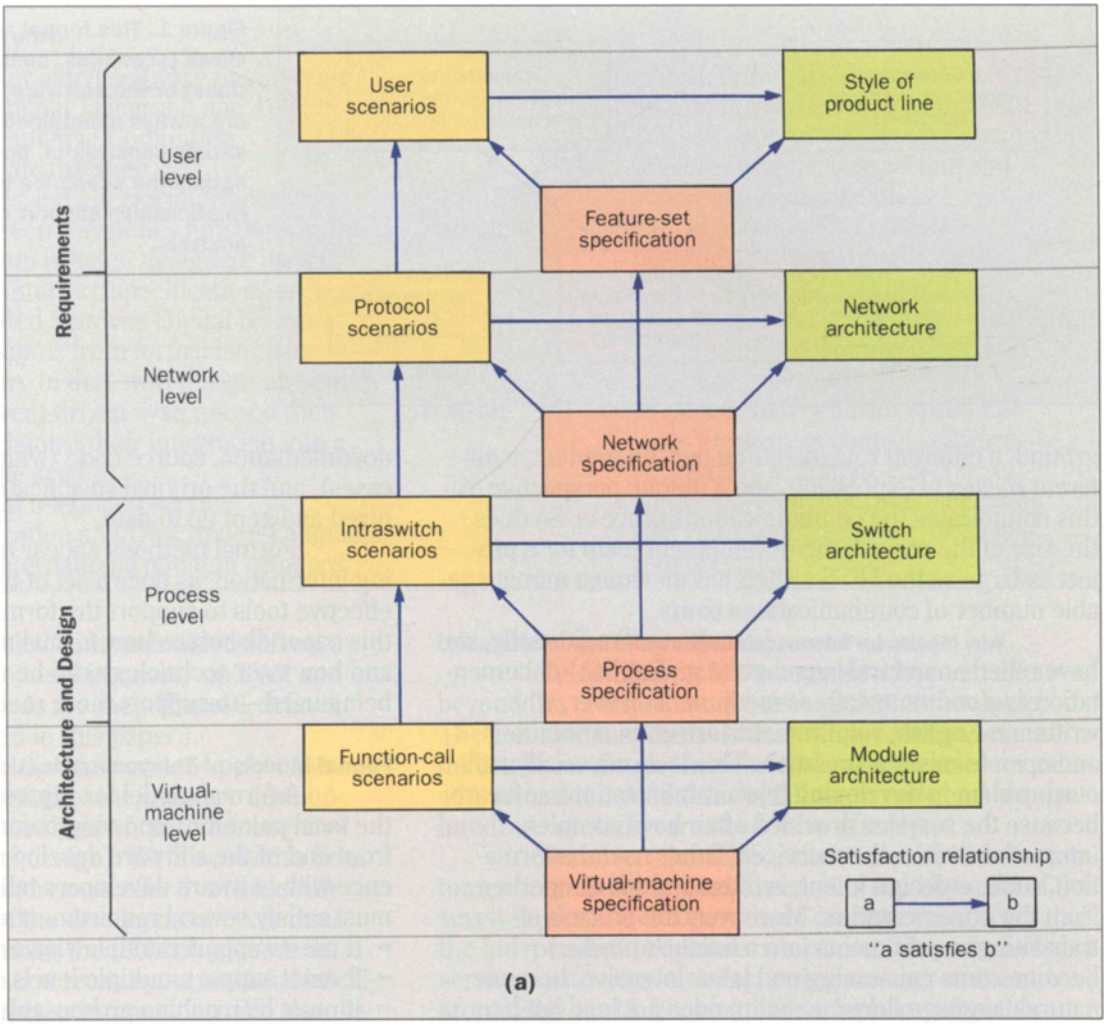


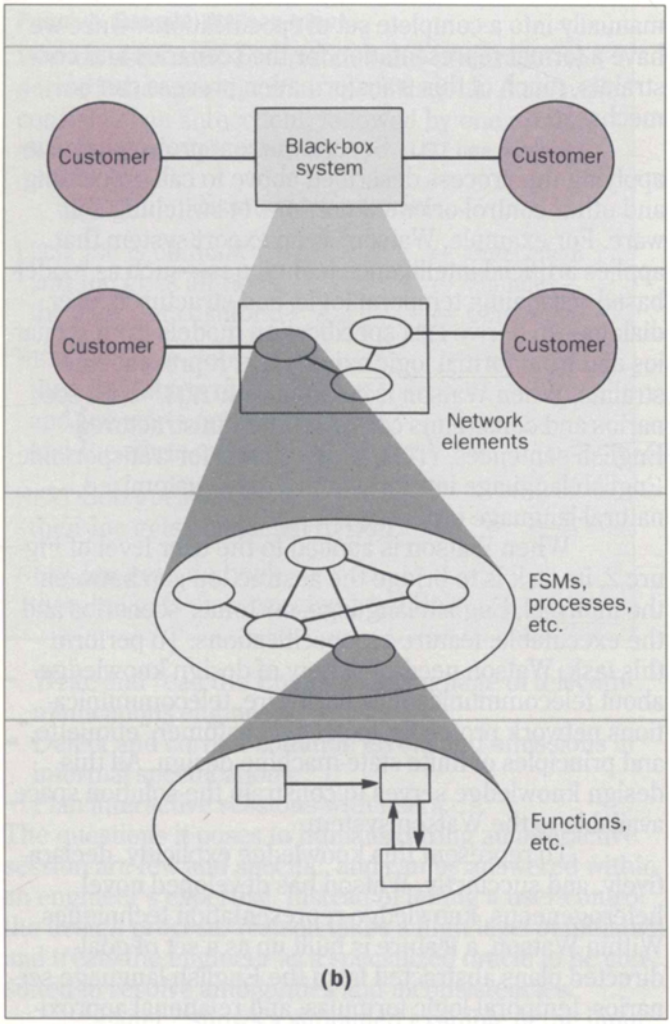
Figure 2. The model can be extended into multiple levels of abstraction. (a) Satisfaction relationships always exist and are maintained between levels, as well as within levels. The user (or customer) level has the highest level of abstraction, while the virtual-machine level has the lowest. (b) As we move to lower levels of abstraction, the higher level's scenarios, constraints, and specifications are elaborated; simplifying assumptions are removed. For example, the black box that a customer sees becomes an interconnected network of telephones and switches; the constraints now include the topology of the network interconnections.

product line (e.g., all special-services access codes begin with a star "*").

- *Specifications*—a complete description of the software. A specification might be a requirements, architecture, or design document.

Satisfaction relationships (identified by the arrows in Figure 1) must always be maintained within these views. That is, scenarios must satisfy constraints, and specifications must satisfy both scenarios and constraints.

Neither the scenarios nor the constraints are complete pictures of the system. But systems engineers typically begin thinking about a new feature in terms of



scenarios and constraints, and then synthesize them manually into a complete specification.

The model also provides for multiple levels of abstraction (see Figure 2):

- *User* level—This is the highest level of abstraction and represents the telecommunications network from a customer's viewpoint.
- *Network* level—This level represents the network's interconnected components.
- *Process* level—This level represents the switching software's many processes, finite-state machines (FSMs), and modules.
- *Virtual-machine* level—This level provides more detailed descriptions of the software elements defined at the process level.

These levels are related by successive elaborations and the removal of simplifying assumptions. As Figure 2a shows, the upper levels can be considered requirements, while the lower levels can be considered design.

To illustrate the levels of abstraction, consider the example in Figure 2b.

As we move from the user level to the network level, notice that we dispense with the simplifying assumption that a telecommunications network is a centralized system. The user level's single, black-box system is replaced at the network level by a network of interconnected telephones and switching systems (the network components). At the network level, the constraints include the topology of network interconnections.

When we move to the process level, we dispense with the assumption that the switching software within one switch can be represented by a single process. Here, the partitioning and connectivity of the various components of the software architecture can offer some of the constraints.

Finally, moving to the virtual-machine level, we dispense with the assumption that call-processing resources (such as Touch-Tone receivers, memory, and message queues) have no limit. We must model what happens when system resources are exhausted.

As we continually elaborate our model, introducing more and more levels of complexity, we add behavioral changes that can propagate up to higher levels. For example, running out of Touch-Tone receivers will introduce an action that a customer will see, even though the action was not originally modeled at the user level.

Note that satisfaction relationships exist between levels of abstraction (e.g., the design must satisfy the requirements, as shown by the arrows in Figure 2a), as well as between views within a single level (e.g., specifications must satisfy both the scenarios and the constraints). When scenarios, constraints, and specifications are represented in machine-processable form, the formal satisfaction relationships form the basis for much of the internal consistency checking and analysis.

In defining the formal models represented in Figure 2, clearly, no single form of representation is well suited to all the views and levels of abstraction that are portrayed. Scenarios, constraints, and specifications are separated in the model precisely because they are different entities. Their separation gives us the opportunity to capitalize on gains that can be achieved by tuning separate representations to each problem.

Thus, one must recognize from the beginning that—in large, complex software systems—multiple representation paradigms and multiple means of expression will be used simultaneously for different parts of the system, for different levels of abstraction, and by different users of the system.[2,3] We believe that this multiparadigm approach, coupled with the ability to compose the different paradigms into one final implementation, is a powerful way to obtain the productivity and quality improvements inherent in domain-specific approaches.

For example, at the user level in Figure 2, scenarios are best represented as structured-English statements or as graphically animated episodes. Constraints are best represented as English-language assertions that can be translated into logic axioms. Some specifications are best represented by sequence diagrams and state diagrams. At another level, call-processing software may best be represented by models of interacting FSMs, while some administrative programs may best be represented by fourth-generation languages.[4]

Process Mechanization Based on the Formal Model

Earlier, we had stated that the systems engineer's job is to transform the scenarios and constraints manually into a complete set of specifications. Once we have a formal representation for the scenarios and constraints, much of this transformation process can be mechanized.

Watson and TELI. Several current projects involve applying the process described above to call processing and other control-oriented portions of switching software. For example, Watson[5] is an expert system that applies artificial-intelligence techniques—such as model-based reasoning; temporal logic; and structured, user dialogs—to derive FSM specification models from scenarios and from formal logic axioms that represent constraints. When Watson is coupled with TELI,[6] both scenarios and constraints can be entered in structured-English sentences. (TELI, which stands for transportable English-language interpreter, is a user-customized, natural-language processor.)

When Watson is applied to the user level of Figure 2, its task is to bridge the abstraction gap between the informal, English language, customer scenarios and the executable, feature-set specifications. To perform this task, Watson needs a variety of design knowledge about telecommunications hardware, telecommunications network protocols, expected customer "etiquette," and principles of finite-state-machine design. All this design knowledge serves to constrain the solution space available to the Watson system.

To represent this knowledge explicitly, declaratively, and succinctly, Watson has developed novel, heterogeneous, knowledge-representation techniques. Within Watson, a feature is built up as a set of goal-directed plans abstracted from the English-language scenarios, temporal-logic formulas, and relational approximations to finite-state machines. The background knowledge and constraints are embedded in prefabricated plans, logic axioms, and language constraints. Theories that interrelate these three basic representation techniques are well known; hence, internal completeness and consistency checking are possible.

Because its users may not be programmers, Watson must be able to:

Panel 2. Sample Watson Input

Watson uses scenario episodes and constraints to derive finite-state-machine specifications. A scenario consists of an antecedent, followed by one or more stimulus-consequence pairs.

Scenario Episodes

FIRST Joe is on hook ← Antecedent
and Joe goes off hook ← Stimulus
then Joe gets dialtone ← Consequence

NEXT Joe dials Bob
then Bob starts ringing
and Joe starts getting ringback
and Joe starts calling Bob

NEXT Bob goes off hook
then Joe gets connected to Bob

NEXT Joe goes on hook
then Joe gets disconnected from Bob

- Write and read the English sublanguage of telecommunications engineers.
- Detect and correct common errors and omissions in informal specifications.
- Plan interactive sessions with users.

The questions it poses to humans during an interactive session are few and specific, and can be answered within an engineer's expertise. Instead of letting a user control the design process, Watson takes a high level of initiative and treats the engineer as a specialized oracle to be consulted to resolve ambiguities and inconsistencies.

Panel 2 shows a simplified example of the input to Watson. The input is a series of episodes, each consisting of one or more stimulus-consequence pairs preceded by an antecedent. The episodes are ordered, and each stimulus results in an assertion about the state, or a change of state, of the call.

In addition to these episodes, constraints are expressed at each level in Figure 2. For example, at the user level, a constraint might be that *a telephone should not ring if it is off hook*. Another constraint may be an assertion about telephone etiquette, e.g., *a telephone should not ring unless it is being called.*

Kaleidoscope. The software system, Kaleidoscope, addresses[7] another problem in producing specifications. Much of the complexity in switching-system software comes from having to deal simultaneously with a multitude of design objectives that become closely intertwined in the final code. Kaleidoscope uses well-known axioms from formal language theory to provide a way to deal separately with design objectives in event-driven systems and then mechanize the integration of these objectives into a single system.

For specifying the control-oriented part of a process, Kaleidoscope uses a highly structured, hierarchical, graphical notation. This language appears to solve the most serious problems of executable specifications by managing nontrivial complexity; allowing automated generation of a readable, English-language version of the specification; providing for automated verification of correctness conditions; and preserving freedom from implementation bias.

This graphical notation currently supports both the structured-diagram representation of FSMs[8] and the Statechart representation.[9] Thus, Kaleidoscope provides a foundation for supporting multiple paradigms. Neither representation alone is adequate for high-quality specification of complex systems, but joining the two under Kaleidoscope provides a powerful combination.

Just as Kaleidoscope can join two complementary FSM representations, it can compose multiple FSM representations of either type into one large FSM. This permits clear separation of design concerns. As an example, consider Figure 3. Here, the *successful* and *failing* POTS (plain old telephone service) calls are specified separately.

As Figure 3a shows, the successful POTS origination is a sequence of *dialing*, *connect*, and *disconnect* parts. In this diagram, the dialing part is further elaborated as a sequence of a *beginning of origination* part

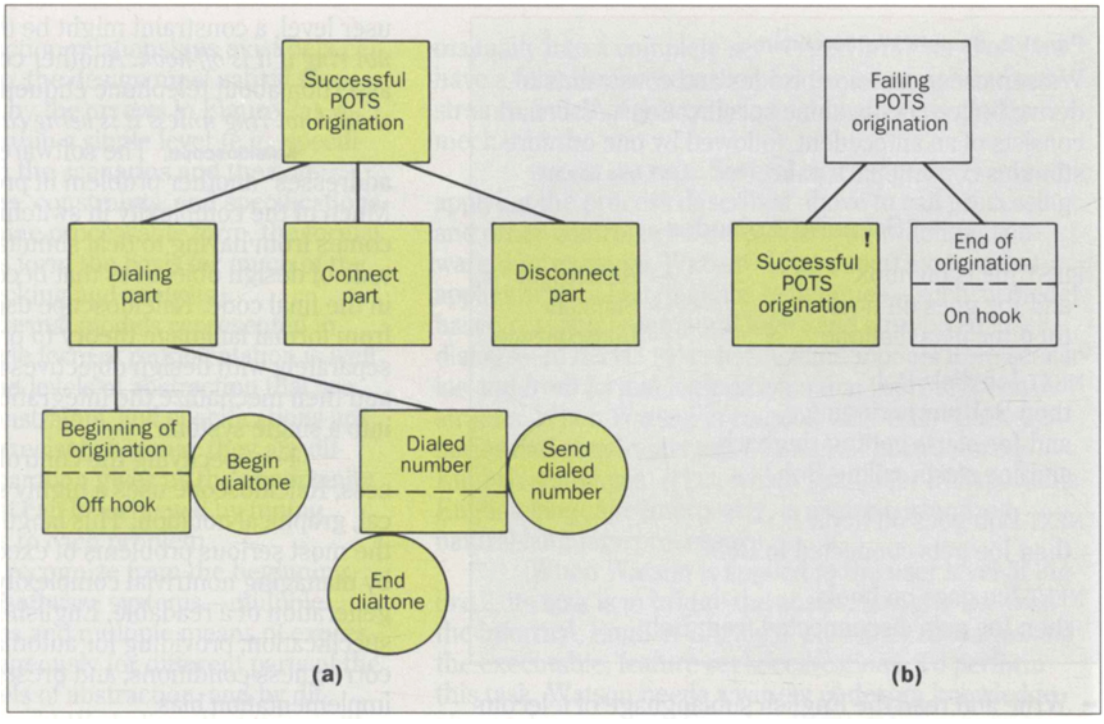


Figure 3. Kaleidoscope allows the two interrelated specifications to be defined separately and combined into one larger specification. Here, we see the result for the interrelated specifications for (a) successful POTS origination and (b) failing POTS origination. The execution order of a specification's subsequences moves from left to right. The "!" block denotes reuse of the subsequence for a successful POTS call.

and a *dialed number* part, with appropriate tones and other outputs generated along the way. In Figure 3b, the *failed POTS call* is specified as any proper subsequence of a *successful POTS call* (denoted by the "!" block), followed by an *end of origination* (on hook) signal.

Obviously, these two specifications must become firmly intertwined in the final code, but there are distinct advantages to keeping them separate at the specification level. For example, the way to abort a call origination could be different for different classes of customers or different kinds of telephones. The rest of the specification could remain unchanged.

The diagram also demonstrates another advantage of the notation: The notation's hierarchical nature displays completeness without requiring complete elaboration at every level. For example, the elaboration of the connect part of the specification can be on a separate page that is headed by the connect-part box. The relationship of the original diagram and the succeeding elaborations is clear without requiring messy interconnections between diagrams.

Kaleidoscope's ability to do composition not only enhances its power, but also makes life simpler for systems like Watson. For example, we can use Watson to specify small FSMs of individual features, knowing that Kaleidoscope will piece them together into one large, complex system.

Inscape. An integrated set of system construction and evolution tools, called Inscape,[10] makes constructive use of formal interface specifications. To support the generation and enforcement of modular specifications, Inscape creates a semantic interconnection model of the

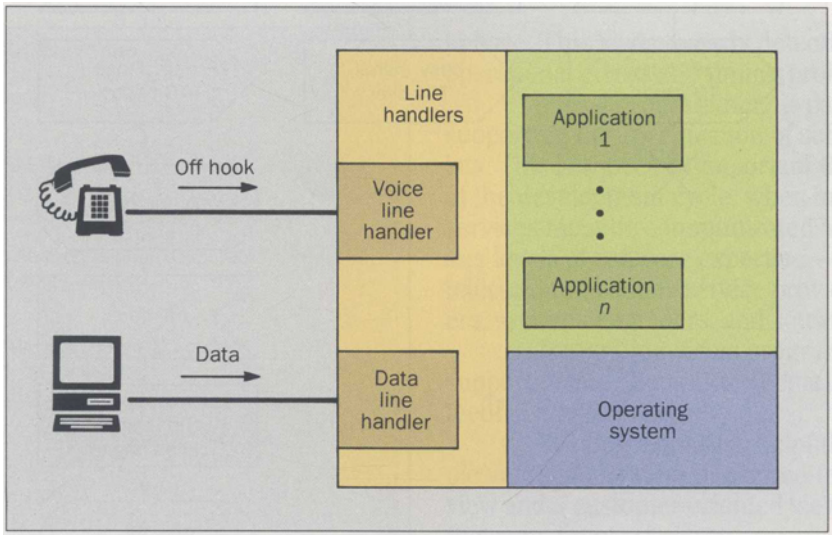


Figure 4. Animated scenarios used in AT&T software for visual specification (SVS) help systems engineers understand the interactions of complex telecommunications features early in the development cycle, thus avoiding costly rework in later phases.

interfaces between functional specifications and their resulting program modules. In particular, Inscape checks that the:

- *Preconditions* are true before a program executes.
- *Postconditions* are true after it executes.
- *Obligations* are satisfied at some point after the execution is completed.

In doing this, Inscape helps software architects compose large systems from small modules in a way that maximizes the possibility of reusing both specifications and software.

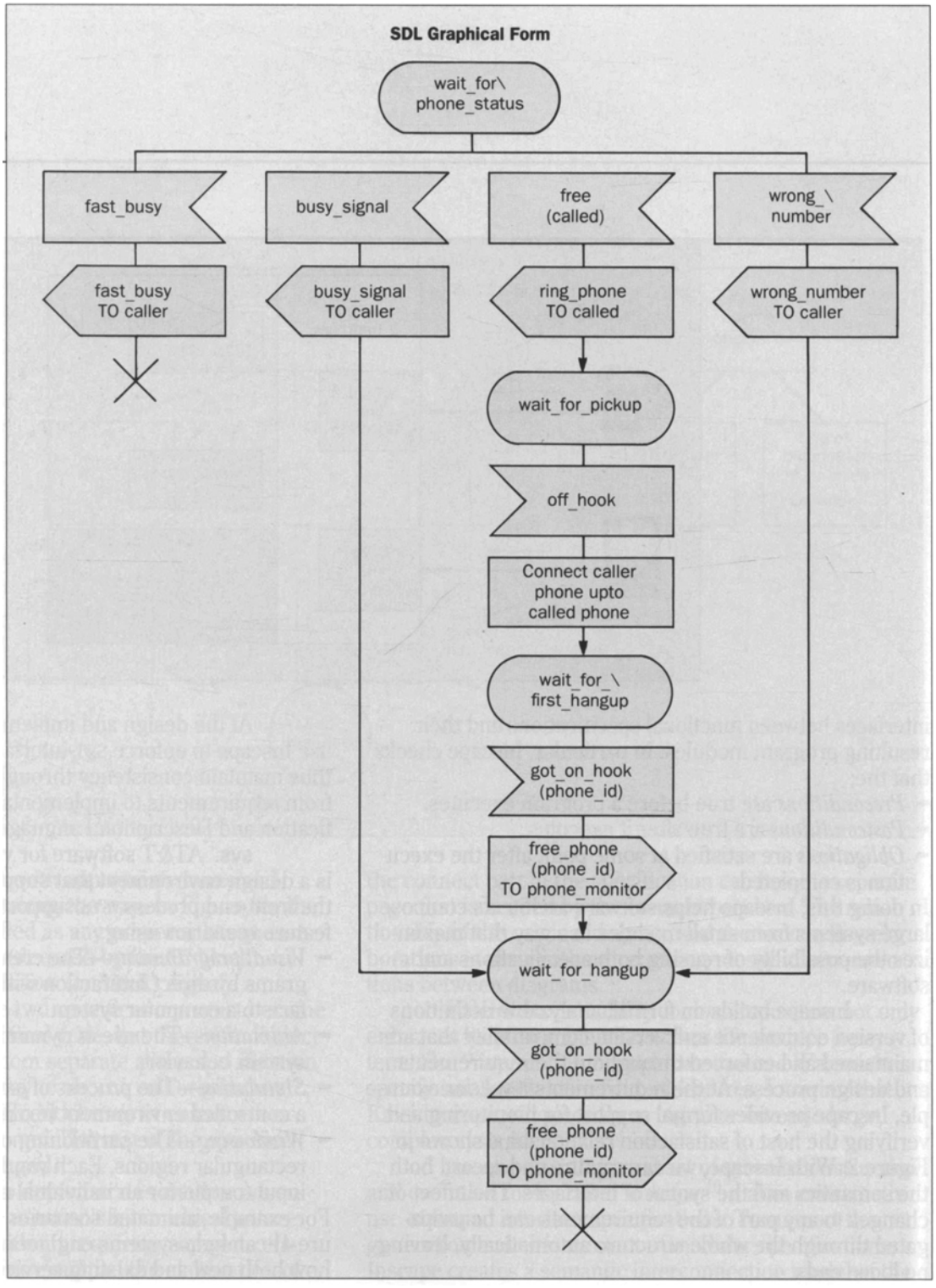
Inscape builds on formal, analyzable definitions of version equivalence and version compatibility that are maintained and enforced throughout the requirements and design process. At the requirements level, for example, Inscape provides formal support for monitoring and verifying the host of satisfaction relationships shown in Figure 2. With Inscape, we can capture and record both the semantics and the syntax of interfaces. The effect of changes to any part of the requirements can be propagated through the whole structure automatically, leaving no loose ends.

At the design and implementation level, we could use Inscape to enforce SDL-interface specifications and thus maintain consistency throughout the whole process from requirements to implementation. (SDL is the Specification and Description Language, described below.)

SVS. AT&T software for visual specification (SVS) is a design environment that supports mechanization of the front-end process. SVS supports the initial creation of feature scenarios using:

- *Visual programming*—The creation of executable programs by direct interaction with the graphical interface to a computer system.
- *Animation*—The use of dynamic graphics to portray system behavior.
- *Simulation*—The process of exercising the scenario in a controlled environment to observe system behavior.
- *Windowing*—The partitioning of a display screen into rectangular regions. Each window supports the input/output for an individual computer program.

For example, animated scenarios (like the one in Figure 4) can help systems engineers and developers see how both new and existing services are supposed to


SDL Graphical Form
wait_for\
phone_status
fast_busy
busy_signal
free
(called)
wrong_\
number
fast_busy
TO caller
busy_signal
TO caller
ring_phone
TO called
wrong_number
TO caller
wait_for_pickup
off_hook
Connect caller
phone upto
called phone
wait_for_\
first_hangup
got_on_hook
(phone_id)
free_phone
(phone_id)
TO phone_monitor
wait_for_hangup
got_on_hook
(phone_id)
free_phone
(phone_id)
TO phone_monitor

SDL Text Form

```
STATE wait_for_phone_status;
    INPUT fast_busy;
        OUTPUT EXTERNAL fast_busy TO caller;
        STOP;
    INPUT busy_signal;
        OUTPUT EXTERNAL busy_signal TO caller;
        NEXTSTATE wait_for_hangup;
    INPUT free (called);
        OUTPUT EXTERNAL ring_phone TO called;
        NEXTSTATE wait_for_pickup;
    INPUT wrong_number;
        OUTPUT EXTERNAL wrong_number TO caller;
        NEXTSTATE wait_for_hangup;
ENDSTATE wait_for_phone_status;

STATE wait_for_pickup;
    INPUT off_hook;
        TASK 'Connect caller phone upto called phone.';
        NEXTSTATE wait_for_first_hangup;
ENDSTATE wait_for_pickup;

STATE wait_for_first_hangup;
    INPUT got_on_hook (phone_id);
        OUTPUT EXTERNAL free_phone (phone_id) TO
            phone_monitor;
        NEXTSTATE wait_for_hangup;
ENDSTATE wait_for_first_hangup;

STATE wait_for_hangup;
    INPUT got_on_hook (phone_id);
        OUTPUT EXTERNAL free_phone (phone_id) TO
            phone_monitor;
        STOP;
ENDSTATE wait_for_hangup;
```

Figure 5. Both the graphical (left) and text (right) forms of SDL allow complex telecommunications features to be specified as finite-state-machine models.

behave. This leads to early detection and resolution of operational errors and timing problems.

Visual programming is particularly helpful in supporting the specification of software by nonspecialists. This is especially important during the early stages of the development cycle, when information about new services must be communicated among people with various levels of software expertise—such as customers, telecommunications service providers, account managers, systems engineers, and software developers.

Integrating visual programming and simulation supports rapid changes to scenarios by giving instant feedback to SVS users.

Windowing is also helpful in that it permits multiple views of a system or service (e.g., a network-oriented view and a customer-oriented view) to be presented at the same time.

In addition to its communications function, SVS can provide a user interface to the completeness and consistency checking operations that Watson, Kaleidoscope, and Inscape perform. In particular, an SVS scenario could be created in an interactive mode, and Watson could provide immediate feedback if actions in the scenario were inconsistent with those specified previously. Watson could also warn the user about things that were not specified completely and about potential feature interactions.

SDL. Watson, Kaleidoscope, Inscape, and SVS are recent technologies. One technology that the 5ESS switch's developers have been using for some time is SDL, a language for the specification and development of telecommunications systems. SDL was developed by the CCITT (International Telegraph and Telephone Consultative Committee), a standards-setting body of the United Nations. Because SDL is an international standard, many buyers of the 5ESS switch around the world require SDL specifications for the switch software they purchase.

SDL implements an extended FSM model of the software structure. The model is ideal for use in call processing, human-machine interfaces, and other situations

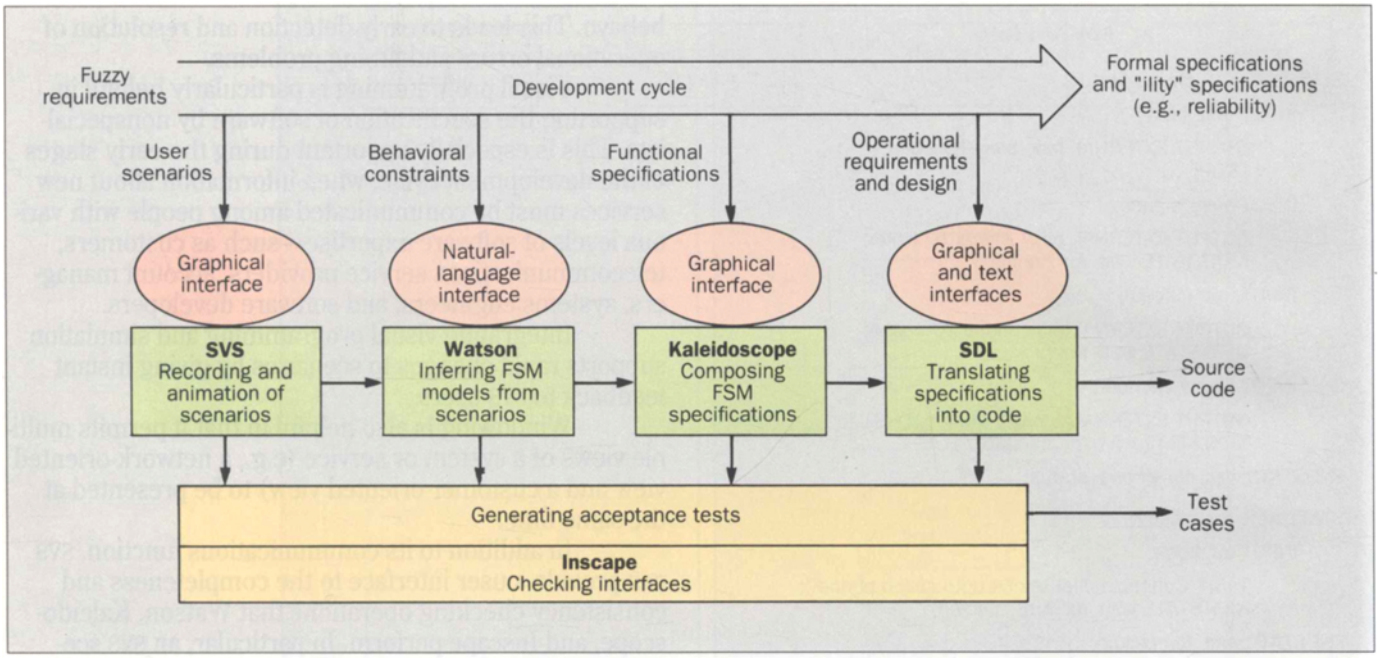


Figure 6. CASE tools and technologies may be integrated to support the front end of the software development process. The development cycle proceeds from left to right in this diagram. The interfaces used earlier in the cycle tend to be more human friendly and more intelligent.

where the software must respond to a variety of stimuli. The language has two forms (Figure 5): a programming-style text version (SDL/PR) and a graphical, flowchart-like version (SDL/GR). The graphical version is one of SDL's most important features, because many developers find pictures and diagrams much easier to understand than text—particularly when the diagrams use standard notation and are based on a formal, well-defined structure.

Systems engineers and software developers on the 5ESS switch team use SDL to describe a system's requirements and progressively refine the description during design and implementation. At the lowest level, SDL specifications are translated automatically into C-language code.

An extensive set of tools supports SDL:

- A *graphical editor* enables systems engineers and developers to create, modify, store, and retrieve SDL/GR representations of their software.
- *Translators* convert between the SDL/GR and SDL/PR versions, and translate the SDL/PR version into C-language code.
- *Analysis tools* check for deadlock, completeness, and consistency of specifications. Thus, basic flaws in requirements and design can be detected *before* any code is generated.
- A *simulator* is available to "execute" specifications and thus expose other defects early in the development process.

- *Report generators* provide additional information, such as cross-reference tables, to assist in requirements analysis and design.

Putting the Pieces Together

By discussing pairwise interactions of the various CASE tools and technologies throughout this paper, we've hinted at how the various pieces could fit together. Figure 6 shows how they might be integrated.

As we proceed in the development cycle (from left to right, in Figure 6), we begin to formalize those "fuzzy requirements" that we know how to process, leaving behind only the more nebulous "ility" specifications (such as reliability and extensibility). Furthermore, a user is offered a choice of input languages, with those earlier in the process providing a friendlier and often, a more intelligent interface.

At the moment, the tools described in Figure 6 have only been integrated on paper. That is, a real 5ESS-switch feature is being "walked through" the diagram, so we can better understand the components and their interfaces. By working with a real feature, we expect to obtain a good understanding of users' requirements for an integrated CASE environment.

A fundamental assumption in the mechanization of the software development process is that test cases will be generated from the specifications and used to verify that the source code is consistent with the specifications. As mentioned earlier, this provides the motivation developers require to keep specifications up to date throughout the process. Work is still needed to determine how test-case generation can best be integrated with the other components.

Strategies for Success

The new software technology described here is, by itself, not enough to meet the challenge posed by large, software development projects without at least three other ingredients:

- A hardware development *platform* to support the software technology
- A process to support *development* of the new software technology
- A process to support *introduction* of the new software technology.

Hardware Platform. An optimal hardware development platform has several key characteristics. The platform:

- Provides an evolution path from previous development platforms
- Provides its users with some advantage over previous platforms
- Provides a clear evolution path to the next generation of platforms
- Supports software from a variety of vendors
- Is cost effective.

Certain other characteristics of the optimal platform are useful to keep in mind:

- *Networking*—A key aspect of a solution is being able to collect various technologies in such a way that they all "talk to each other."
- *Bandwidth*—Most of the CASE applications today are graphically oriented. The ability to support graphics requires certain bandwidth capabilities.
- *Software*—The hardware technology must support the desired software packages for the front-end process.

Given all the above characteristics, the hardware platform should include terminals, workstations, and servers linked together in a local-area or wide-area network of adequate bandwidth per terminal [at least 10 Mb/s (megabits per second)].

Development of New Technology. As stated earlier, a process to support development of the new software technologies is the second key ingredient for success.

To meet the challenge posed earlier in this paper, we require expertise in several different areas. Thus, an interdisciplinary team is essential. At AT&T Bell Laboratories, a division of AT&T, organizations involved in soft-

ware engineering research, exploratory development, software development systems, and quality assurance have been working together and with the 5ESS switch's systems engineers and developers to craft a solution. Organizations on other large projects are also working with the 5ESS switch team to share technology and potential solutions, because many of the same problems are found in other large projects, as we stated earlier.

The importance of a human-friendly interface in the proposed solution also suggests an iterative development methodology, based on the prototyping of new tools and on feedback from customers who test the tools. This methodology has already been used successfully in the current software development environment for the 5ESS switch.

Introduction of New Technology. The motivation for improving the front-end process comes from studies of the current development process and from identifying opportunities for improvement. However, the process improvements need to be engineered properly and integrated into the existing development environment. These process improvements must also be supported by adequate training and documentation and by a measurement scheme to assess their value.

One must also remember that new services are being added to an existing system, and that the software to implement these new services must always interface properly with the existing software. When an army of programmers develops a system over several years without the aid of graphical tools, understanding the existing software can be difficult and time consuming. While the technologies described here leave an accurate, graphical, easy to understand description of *new* software, other technologies are needed to help developers understand *existing* software before they modify it to provide new services.

Summary and Conclusion

In this paper, we have discussed some of the challenges inherent in supporting the development of software for large, complex, real-time systems, such as AT&T's 5ESS switch. We believe that many of these challenges are really communications issues, and relying on natural-language documentation as a communications medium is not enough to meet that challenge. What is needed, instead, is a new development process with a formal way to capture information at the front end, and an integrated set of tools to process that information.

We have reported on some of the work being undertaken to meet these challenges. We have concentrated on software technology—CASE tools, in particular. Some of the tools described here are in widespread use within AT&T; others are just coming out of the research laboratory. When all these tools are fully rolled out and integrated, systems engineers and software developers will rely on graphical techniques to describe new services and understand existing services. Thus, requirements, architecture, and design will be described in a much clearer, more consistent way and maintained on line as "living" documents. Completeness and consistency checking, as well as documentation and test-case generation, will be substantially mechanized.

The technology looks promising, but technology is only part of the answer. Improving the front-end process, indeed improving *any* process, requires that management, organizational, and training issues also be addressed.

Acknowledgments

We would like to thank the following individuals who have also contributed to this paper: Al Bengtson, Eric Beyler, Van Kelly, Vickie Klick, Bob Kosman, Dewayne Perry, Lai-Cherng Suen, Tim Surratt, and Pamela Zave.

References

1. M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 182-211.
2. P. Zave, "A Compositional Approach to Multiparadigm Programming," *IEEE Software*, Vol. 6, No. 5, September 1989, pp. 15-25.

3. D. Perry, "Industrial Strength Software Development Environments," *Information Processing '89: The IFIP World Congress Proceedings*, San Francisco, California, August 28 to September 1, 1989, G. X. Ritter (ed.), North-Holland Publishing, New York, 1989, pp. 195-203.
4. J. Martin, *Fourth Generation Languages*, Savant Research Studies, Carnforth, Lancashire, England, 1983.
5. V. Kelly and U. Nonneman, "Inferring Formal Software Specifications from Episodic Descriptions," *Proceedings AAAI-87: Sixth National Conference on Artificial Intelligence sponsored by The American Association for Artificial Intelligence*, Seattle, Washington, July 13 to 17, 1987, Volume 1, Morgan Kaufmann Publishers, Los Altos, California, 1987, pp. 127-132.
6. B. Ballard, "A Lexical, Syntactic and Semantic Framework for TELI: A User Customized Natural Language Processor," *Relational Models of the Lexicon*, Martha Evens (ed.), Cambridge University Press, Cambridge, England, 1988, pp. 211-236.
7. P. Zave and D. Jackson, "Practical Specification Techniques for Control-Oriented Systems," *Information Processing '89: The IFIP World Congress Proceedings*, San Francisco, California, August 28 to September 1, 1989, G. X. Ritter (ed.), North-Holland Publishing, New York, 1989, pp. 83-88.
8. M. A. Jackson, *System Development*, Prentice-Hall International, Englewood Cliffs, New Jersey, 1983.
9. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. VIII, 1987, pp. 231-274.
10. D. Perry, "The Inscape Environment," *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, Pennsylvania, May 15 to 18, 1989, Association for Computing Machinery, Washington, D.C., 1989, pp. 2-12.

Biographies (continued)

from the University of Michigan and an M.S. from the University of Wisconsin at Madison, both in mathematics. Mr. Smith is involved in quality management of the software development process for the 5ESS switch. He joined the company in 1968 and has a B.S.E.E from Brigham Young University, and both an M.S.E.E and a Ph.D. in electrical engineering from Stanford University. Mr. Zislis is responsible for exploration of advanced software architectures, development environments, process modeling, specifications support, and graphics. He joined the company in 1969 and has a B.S. in mathematics and computer science from the University of Illinois, an S.M. in information sciences from the University of Chicago, and a Ph.D. in computer sciences from Purdue University.

(Manuscript received October 26, 1989)