A SIMPLIFIED GRAPHIC NOTATION FOR Ada PROGRAMS

D. Sterne[1]
A. Glendening
B. Jachowski
G. Pretti

GTE Government Systems,
Rockville, Maryland

ABSTRACT

A number of graphic notations for describing Ada programs have been proposed. In general, these are not supported by off-the-shelf tools; they are perceived by nonexperts as incomprehensible; they obscure simple design deficiencies; and they cannot adequately represent design overviews. To address these problems, a simplified graphic notation has been developed and used in a large Ada application at GTE Government Systems Corporation, in Rockville, Maryland. The notation consists of three interrelated diagrams based on extensions of structure chart concepts.

INTRODUCTION

Unlike earlier widely-used programming languages, Ada provides a variety of building blocks from which programs can be constructed, including packages, tasks, subprograms, and generics. Consequently, the topology of Ada programs is complex [BOOCH] and not readily represented by simple traditional diagrams such as dataflow diagrams [DEMARCO], structure charts [MYERS], and block diagrams. New, more descriptive diagrams, tailored for Ada, have been developed [BOOCH] [BUHR] [CHERRY] [RATIONAL] [BURKH] [NIELS] [STERNE85]. These diagrams tend to be significantly more complex than traditional diagrams, however, and most present one or more of the following drawbacks.

A. Until recently, few released commercial off-the-shelf tools have existed to support the drawing or automated analysis of these diagrams, except for general-purpose graphical editors such as the MacDraw program for the Apple Macintosh computer [MACDRAW].

B. The diagrams may be viewed as incomprehensible by senior engineers and customers because their interpretation requires an extensive understanding of Ada constructs. Although persons of limited Ada expertise may be unable to critique the details of an Ada design, those having extensive software engineering experience can contribute valuable feedback concerning the organization and architecture of large programs. However, without suitable graphic representations, these individuals are effectively excluded from the design review process. (Recall that most fundamental principles of good software design, including hierarchical structure, information

---

[1]**Daniel F. Sterne** *may be contacted at Trusted Information Systems in Glenwood, Maryland*

hiding, high module cohesion and low coupling, and disciplined use of concurrency, predate Ada and are language independent.)

C. The diagrams may represent so much detailed information, such as individual task entries, that they are unsuitable for portraying a system-level overview.

D. The diagrams may attempt to show too many independent dimensions at once, including containment, visibility, concurrency, control flow, and data flow. As a result, simple design deficiencies, such as circular dependencies among tasks or among packages [NISSEN] [DEC] [STERNE86], may be obscured. Although in principle, sophisticated tools might be able to detect these deficiencies in complex diagrams, the importance of intertask and interpackage relationships dictates that they be conspicuously depicted for human designers and design reviewers.

E. The diagrams may include informal annotations, such as clouds, causing them to be insufficiently rigorous or standardized to be used as formal deliverable documentation, especially for large software projects.

At GTE Government Systems in Rockville, MD, a new simplified notation for Ada programs has been developed and used successfully as the primary graphic representation for a several hundred thousand-line Ada application. The notation comprises three types of interrelated diagrams that provide the essential views of program organization needed for design reviews and maintenance. These include two high-level views, called the Withing Diagram and the Tasking Diagram, and a more detailed view called the Ada Structure Chart. All three types of diagrams are based on extensions of traditional structure chart concepts and are constructed from structure chart symbols, to which a few additional semantics are given depending on diagram type.

In its emphasis on simplicity, economy, and separation of concerns, this notation departs from better known and more elaborate graphical approaches, and provides the following benefits.

A. It conveys the essential dimensions of an Ada design independently, effectively, and at the level of detail appropriate to each dimension.

B. Its diagrams are simple enough to have the following properties.

1. They can be understood by customers and senior engineers familiar with traditional block diagrams and structure charts.

2. They can be constructed and maintained using existing off-the-shelf structure chart editors.

3. They can be analyzed by simple, low-cost automated tools, and easily standardized for formal documentation of large systems.

OVERVIEW OF NOTATION

A traditional structure chart [MYERS] shows the calling relationships among subprograms. In addition, it shows relative levels of responsibility and abstraction via placement of symbols along the vertical axis. The advantages of structure charts include simplicity, usefulness, ease of understanding, and familiarity to many software engineers and customers. For large Ada applications, however, structure charts have two major disadvantages.

A. The subprogram-level view provided by structure charts is too low, in terms of level of abstraction, to represent the architecture of a large program. Discerning the organization of a program containing several thousand subprograms by examination of structure charts is a difficult task at best.

B. Ada provides many important constructs and relationships not readily described by structure charts, including tasks and entries, generics, packages, compilation dependencies, elaboration, visibility, lexical nesting, and concurrency.

GTE's approach to the first problem is employ scaled-up summary structure charts to provide a "big-picture" view. Since Ada programs have two important dimensions, compile-time structure and run-time structure, two different summaries are needed: the Withing Diagram and the Tasking Diagram. A means to navigate between these two representations is also provided.

GTE's approach to the second problem is to adapt traditional structure chart notation to encompass several Ada-specific constructs. The resulting diagrams, called Ada Structure Charts, are used primarily to represent the internals of library packages.

THE WITHING DIAGRAM

A Withing Diagram represents a summary of the static organization of an Ada program's source code. Conceptually, it is a high-level structure chart in which each box represents a group of subprograms (and tasks) rather than individual subprograms. More precisely, a box on a Withing Diagram represents a library unit group or LUG as defined in [GRAU], that is "a single Ada library unit, its body (if any), all subunits of the library unit, and anything nested within the library unit, the body, and the subunits." A Withing Diagram treats the four types of LUGs - package LUGs, subprogram LUGs, generic unit LUGs, and generic instantiation LUGs - as equivalent entities.

As shown in figure 1, arrows between boxes represent compilation dependencies between LUGs. The arrow drawn from the LUG DEVICE_MGMT_PKG to the LUG STATUS_COORDINATION_PKG means that some part of DEVICE_MGMT_PKG contains the Ada clause "WITH STATUS_COORDINATION_PKG;" and is dependent on that unit. Frequently, a compilation dependency is analogous to the calling relationship shown by an arrow on a traditional structure chart. However, an arrow on a structure chart represents a call from a single subprogram to another single subprogram, whereas an arrow on a Withing Diagram can represent calls from many subprograms (or tasks) in one LUG to many subprograms (or tasks) in another. Compilation dependencies may also be caused by access to visible declarations of types, constants, objects, or exceptions, or to generic units for purposes of instantiation.

A consequence of depicting all compilation dependencies is that widely-referenced packages of declarations (e.g., global data types) or utility subprograms will appear on a Withing Diagram as connected to many other LUGs. This may obscure other more significant dependencies. To avoid this problem, GTE conventions allow such extensively "withed" LUGs to be enumerated on a "waiver list" and omitted from a Withing Diagram. Beyond being a notational convention, the waiver list is a specification fed into automated diagram analysis and construction tools, as described in a later section.

Since a large program may consist of many more LUGs than can be shown on a one-page diagram, a diagram can be split into multiple

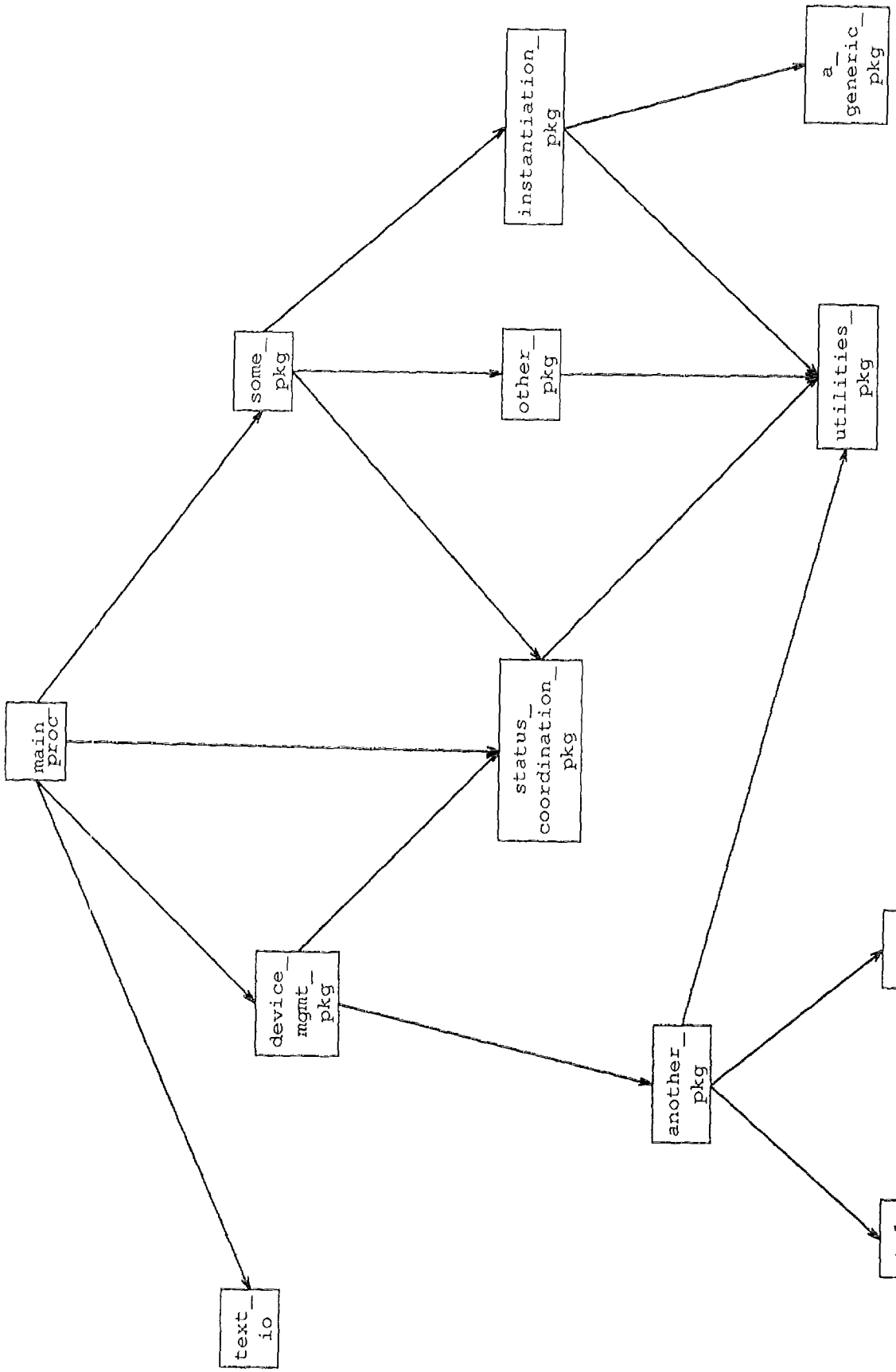

Figure 1. Example Withing Diagram. Boxes represent library unit groups.

pages using page connector symbols. Nevertheless, a single-page Withing Diagram can typically depict as many as 15 to 25 LUGs. Assuming use of identical character size and line widths, a Withing Diagram can provide a 5-fold to 10-fold, or greater, improvement in "field of view" over better-known notations, such as those proposed in [BOOCH] and [BUHR]. The degree of improvement depends on the average number of contained units and visible declarations per LUG. In such better-known notations, each unit or declaration may require its own labeled rectangle, parallelogram, oval, or other symbol, thereby occupying page space that could otherwise be used for additional LUGs. The Withing Diagram's wider field of view can be invaluable when a higher-level design perspective is needed, especially during development of large programs.

GTE conventions require that the boxes on a Withing Diagram be arranged so that all arrows point downward. This serves several purposes. First, this forces a top-down view of the program, in which more abstract functions are at the top, and more detailed functions are at the bottom. Second, it facilitates a rapid initial assessment of the potential impact of changing or recompiling any LUG; any other LUG encountered by tracing upward from the changed LUG through the diagram arrows may potentially be affected. (For example, figure 1 shows that the only LUG potentially affected by a change in DEVICE_MGMT_PKG is the main procedure MAIN_PROC.) Third, it reveals circular dependencies among LUGs, because it is impossible to arrange the boxes as required, if a circularity exists. In general, violatons of this convention are obvious to the human eye, and for this reason, the current automated tool set for Withing Diagrams does not analyze this property.

Although Ada permits circular dependencies among LUGs, for the following reasons, GTE design conventions do not.

A. If a circularity exists among a set of LUGs, each LUG depends directly or indirectly on all others in the set. This impedes reusability because no single LUG or subset of LUGs can be used without modification in a different context.

B. If a circularity exists among a set of LUGs, bottom-up testing cannot be employed, because there is no "bottom-most" unit that can be tested by itself and then used in testing of "higher" units; testing for each LUG requires creation of a stub for at least one other LUG.

C. If a set of LUGs contains tasks that communicate across LUG boundaries, the default elaboration order chosen by the compiler may result in exceptions being raised during elaboration; namely, a task in one LUG may attempt to rendezvous with a task in another LUG whose body has not yet been elaborated. If this occurs, the programmer must explicitly constrain the order of elaboration through use of pragma elaborate. If, however, a circularity exists among such a set of LUGs, determining an error-free elaboration order may require substantial analysis by the programmer. By contrast, in the absence of circularities, no analysis is required; any bottom-up elaboration order will suffice, and can be achieved mechanically, by adding pragma elaborate statements to parallel each existing context clause in the offending units. In addition, since aspects of elaboration order are implementation dependent, the absence of exceptions during elaboration for one Ada compiler does not preclude the occurrence of exceptions for others. Therefore, the ability to employ pragma elaborate mechanically, afforded by avoiding circularities, may be valuable as a defensive measure for portable applications.

THE TASKING DIAGRAM

The Tasking Diagram summarizes the run-time overview of an Ada program, showing the concurrency, control flow, and primary data flow among Ada tasks. Like a Withing Diagram, it is conceptually a high-level structure chart, in which boxes on the diagram represent subprogram groups rather than individual subprograms. However, instead of grouping subprograms together according to LUG membership, subprograms are combined into task-centered groups according to calling chains. Each box on a Tasking Diagram represents an Ada task together with all subprograms it calls, directly or indirectly as a result of chained subprogram calls. By definition, a subprogram call chain ends once it reaches a task entry. Since the main program, which is called by the "environment task", behaves as another concurrent thread of execution [ARM], it too, together with the subprograms it calls, is represented by a box. External entities that may affect the concurrency of tasks, such as I/O devices and operating system components, are represented by double-sided boxes.

Since subprograms do not affect concurrency, they are not shown explicitly but are included implicity in the boxes representing the tasks that call them. Similarly, since package boundaries do not affect concurrency, packages are not shown graphically. Nevertheless, the names of packages that contain tasks do appear, but only as parts of box labels, which are given in <package_name>.<task_name> format, allowing convenient navigation between Tasking and Withing diagrams. Names of packages that do not contain tasks, however, do not appear.

As shown in figure 2, the CONTROLLER task in DEVICE_MGMT_PKG calls at least one entry in the COLLECTOR task in STATUS_COORDINATION_PKG; the call may be chained through layers of subprograms. The directed bubble, a "data couple," represents the primary flow of data, which in this case shows output from the CONTROLLER to the COLLECTOR. The flow of data may be input, output, or both. Figure 2 also indicates that the CONTROLLER task calls the COMMAND_BUFFER task in DEVICE_MGMT_PKG and obtains data from it, and communicates bidirectionally with an I/O device.

As for Withing Diagrams, GTE conventions require that the boxes on a Tasking Diagram be arranged so that all arrows (call chains) point downward. This serves two purposes. First, it forces a top-down view of the program, this time from a control-flow standpoint. Arranged this way, a generalized stratification according to level of authority is revealed. The tasks "in charge" will appear at the top, with subordinates near the bottom, and server tasks will appear beneath their customers. Second, it detects circular calling relationships among tasks, because it is impossible to arrange the boxes as required, if a circularity exists. Circular calling relationships among tasks can harbor circular deadlock and should generally be avoided [DEC] [NISSEN].

A Tasking Diagram can reveal and summarize the fundamental run-time organization and purpose of a program. For example, from figure 2, the following observations can be made. Of the six concurrent entities (boxes) shown, the main program and the device controller task are clearly "in charge", because they have subordinates and no superordinates; this is reflected in their vertical positioning with respect to the other entities. The main program fetches input, probably commands, from a terminal, and some form of status, from the status collector task. Based on these inputs, it sends output, to a command buffer. The device controller task retrieves commands from the buffer, manipulates an I/O device in response, and reports the results to the status collector task. The results,

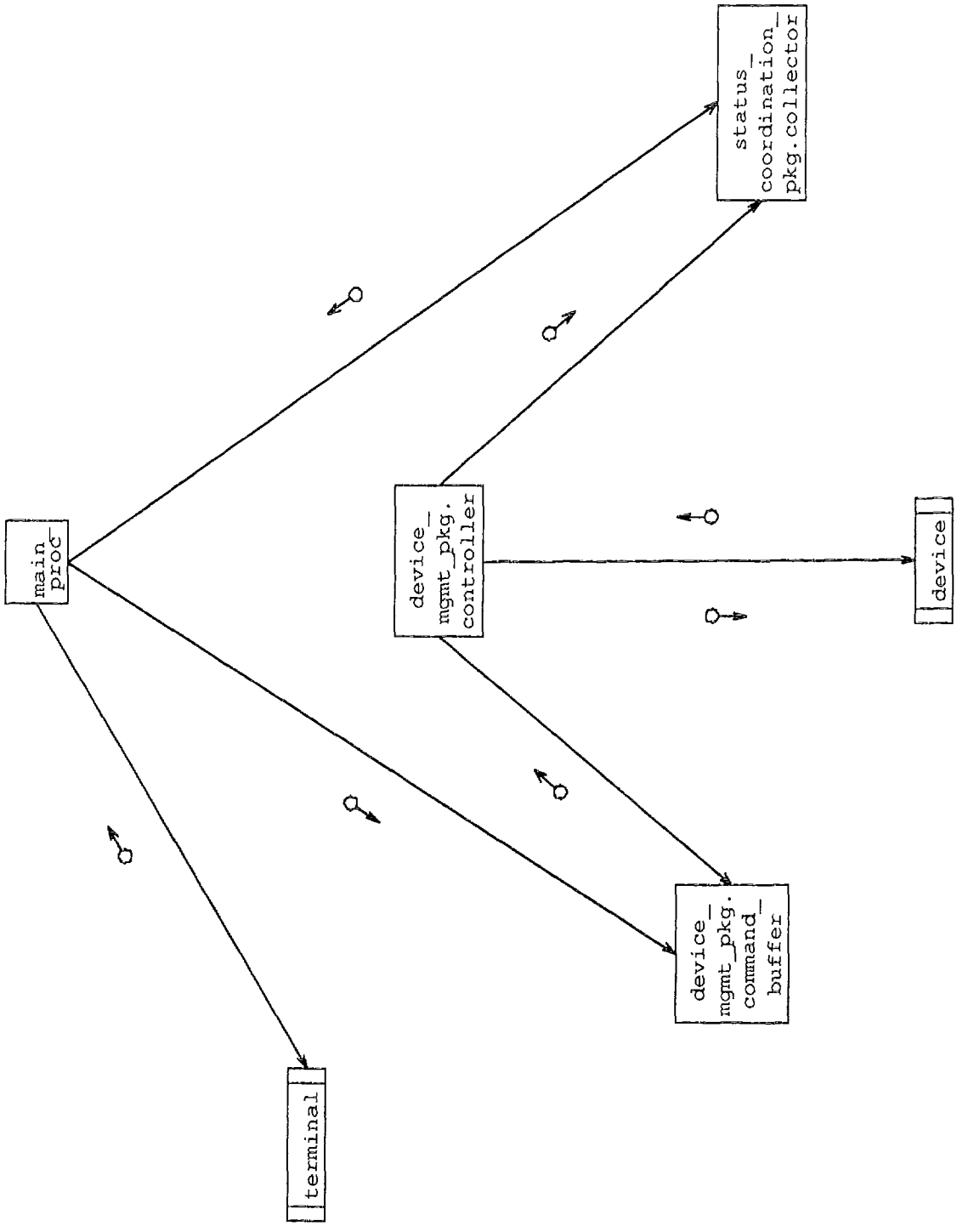

Figure 2. Example Tasking Diagram. Boxes represent concurrent run-time contexts.

or perhaps summaries of previously reported results, are later retrieved by the main program. In short, the program is a closed loop control application having a human operator in the loop. Note that little of the preceding information can be deduced from the Withing Diagram. Note also that if the Withing and Tasking Diagrams were merged into a single diagram, the Tasking Diagram's clear depiction of run-time structure would be obscured by the Withing Diagram's depiction of compile-time structure. This points out the value of representing the compile-time dimension separately from the run-time dimension.

THE ADA STRUCTURE CHART

The Ada Structure Chart is a traditional structure chart [MYERS] showing calling relationships, with additional notational conventions for Ada.

Given a large enough sheet of paper, it is theoretically possible to represent the global call tree for an entire program on a single structure chart. In practice, the call tree must be partitioned into several structure charts. The GTE convention is to partition the global call tree at LUG boundaries, putting each LUG on its own structure chart. A LUG that is too big for a single-page chart is drawn on a multiple-page chart, using page connector symbols as junctions between pages. A different symbol, the double-sided box is used as a junction between the charts belonging to different LUGs.

An Ada structure chart shows a detailed view of a LUG's internal structure and calling relationships, and interfaces to other LUGs. Each box on an Ada structure chart represents a subprogram, task, or package initialization block. Figure 3, shows the Ada structure chart for the package STATUS_COORDINATION_PKG, consisting of a collection of call trees, one for each visible subprogram or task. Some call trees merge because they have common components, such as the hidden task COLLECTOR; others are disjoint. Each tree depicted descends until completed or until it exits the LUG boundary, in which case the first subprogram or task encountered outside the boundary is shown (for example SOME_FUNCT1 in the UTILITIES_PKG LUG). To distinguish between visible and hidden components, hidden components are drawn as single-sided boxes; visible components, even those belonging to other LUGs are drawn as double-sided boxes to indicate their roles as LUG interfaces and chart junctions.

AUTOMATED SUPPORT TOOLS

The three types of diagrams shown here are constructed, edited, and maintained at GTE using the Structure Chart Editor component of the Tektronix Structured Design Tools running on VAX/VMS systems [TEKT]. Diagrams created by this tool are stored on disk as files of character strings that can be easily parsed by diagram analysis tools, thereby extracting titles, labels, symbol types, symbol positions, and connectivity information. At present, two simple automated tools have been developed to augment the Structure Chart Editor. Both support development and maintenance of Withing Diagrams.

The first tool, the Evaluator, compares a Withing Diagram and accompanying waiver list to the withing relationships found in a set of previously compiled Ada units. During the design phase, the diagrams are compared by the Evaluator with Ada specifications and body skeletons. During the implementation and maintenance phases, diagrams will be compared with specifications and fully coded bodies. If changes are introduced into the code after the diagrams have been produced, the Evaluator can be used to identify corresponding diagram changes needed to maintain consistency. It is expected that this capability will contribute

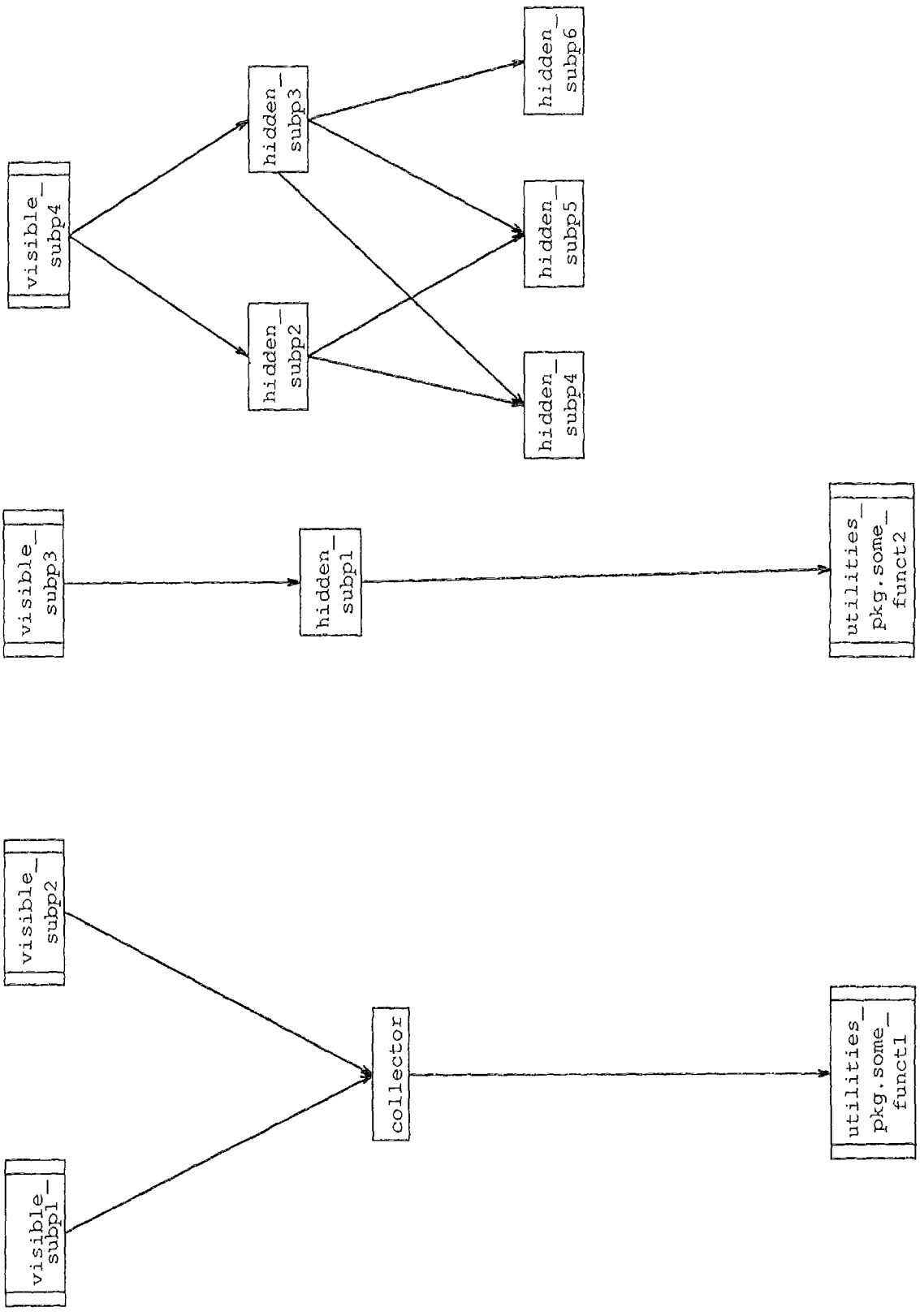

Figure 3. Example Ada Structure Chart showing the internal structure of status_coordination_pkg.

towards keeping design documentation up to date, a historically difficult task.

The second tool, the Builder, produces an "as built" Withing Diagram from a set of compiled Ada units and an accompanying waiver list. This tool is used primarily to produce a first draft of a diagram that can be edited to obtain a more pleasing positioning of symbols. The development of similar pairs of tools for Tasking Diagrams and Ada Structure Charts has been investigated and appears relatively straight forward, especially if these tools make use of symbol table information produced by other tools, such as compilers.

EXPERIENCE TO DATE

Withing and Tasking Diagrams have been used extensively at GTE throughout the preliminary design phase in the development of a several hundred thousand-line Ada application. Ada Structure Charts have been used for portions of the application that have proceeded into the detailed design phase. During both internal and customer design reviews, these diagrams have served, with excellent results, as the primary design "road map" through sets of compiled Ada specifications.

Preliminary experience using the Withing Diagram Builder and Evaluator tools has been positive, and suggests that these tools will provide a convenient and effective means of maintaining consistency between Withing Diagrams and code throughout the software lifecycle. These tools also demonstrate the ease with which other diagram analysis tools may be developed.

SUMMARY AND CONCLUSION

A simplified graphic notation for representing the structure of Ada programs has been described. This notation is being used extensively at GTE Government Systems in the development of a large Ada application. In its emphasis on simplicity, economy, and separation of concerns, this notation departs from better known and more elaborate graphical approaches. Experience to date indicates that the notation provides the following benefits.

- It effectively describes the essential dimensions of an Ada design.
- It is understandable by persons having limited Ada expertise.
- It is compatible with existing off-the-shelf structure chart editors.
- It is conducive to automated analysis by simple tools.

REFERENCES

[ARM] Reference Manual For the Ada Programming Language, ANSI/MIL-STD-1815A-1983, United States Department of Defense, February, 1983.

[BOOCH] Booch, G., Software Engineering with Ada, Benjamin/Cummings, Menlo Park, CA, 1983.

[BUHR] Buhr, R.J.A., System Design with Ada, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[BURKH] Burkhardt, B., Lee, M., "Drawing Ada Structure Charts," ACM Ada Letters, VI, 3, May 1986, 71-80.

[CHERRY] Cherry, G.W., The PAMELA Designer's Handbook, Thought**Tools, Inc., Reston, VA, 1986.

[DEC] VAX Ada Programmer's Run-time Reference Manual, AA-EF88A-TE, Digital Equipment Corp., Maynard, MA, Feb 1985, 7-18.

[DEMARCO] DeMarco, T., Structured Analysis and System Specification, Yourdon Press, New York, NY, 1975.

[GRAU] Grau, J.K., Gilroy, K.A., "Compliant Mappings of Ada Programs to the DOD-STD-2167 Static Structure", ACM Ada Letters, VII, 2, 1987, 73-84.

[MACDRAW] MacDraw, Apple Computer, Inc., Cupertino, CA, 1984.

[MYERS] Myers, G.J., Composite Structured Design, Van Nostrand Reinhold, New York, NY, 1978.

[NIELS] Nielsen, K.W., Shumate, K., "Designing Large Real-Time Systems With Ada," Communications of the ACM, 30, 8, August 1987, 695-715.

[NISSEN] Nissen, J., Wallis, P., Portability and Style in Ada, Cambridge University Press, Cambridge, UK, 1984, 157.

[RATIONAL] "Large System Development and Rational Subsystems," Doc. No. 6004, Rational, Mountain View, CA, November, 1986.

[STERNE85] Sterne, D.F., et al, "Use of Ada for Shipboard Embedded Applications," Proceedings of the Washington Ada Symposium, Laurel, MD., March 24-26, 1985.

[STERNE86] Sterne, D.F., et al, "Package Coupling and Hierarchial System Structure in Ada," Johns Hopkins University Applied Physics Laboratory CER-86-002, March 1986.

[TEKT] Tektronix Inc., "Structured Design Tools User's Manual for VAX/VMS Hosts," March 1986.