3-1984

View-3 and Ada: Tools for Building Systems with Many Tasks

Ann Kratzer
Dartmouth College

Mark Sherman
Dartmouth College

VIEW-3 AND ADA: TOOLS FOR BUILDING SYSTEMS WITH MANY TASKS

Ann Kratzer, Mark Sherman

Technical Report PCS-TR86-117

View-3 and Ada: Tools for Building Systems with Many Tasks

Ann Kratzer
Thayer School of Engineering

Mark Sherman
Department of Mathematics and Computer Science

Dartmouth College
Hanover, NH. 03755

Abstract

This paper discusses some useful features for tools that are intended to be used for developing systems with multiple tasks. We include a description of one tool that has been build, View-3. We also describe some problems that might be encountered when trying to fit this kind of tool into an APSE system.

I. Introduction

The tools required by a project team vary with the kinds of projects they are working on. Projects using Ada[1] frequently build embedded applications which typically use multiple processes or processors. These projects, therefore, need tools to assist project members in developing and testing programs that contain large numbers of interacting Ada tasks. Unfortunately, there is relatively little experience with such tools, so a manager responsible for acquiring tools for specifying and debugging multiple processes may not have a clear understanding of what should be provided. To assist a manager in procuring such tools, we briefly describe one such tool developed at Dartmouth College, our experience with this tool, how such a tool might be integrated in an APSE and what a planner should look for in acquiring similar aids.

II. The View-3 System

The View-3 system is an integrated environment in that it provides facilities for describing and creating processes, for wiring processes together, and for monitoring process execution. After summarizing each of these aspects, this section gives an overview of the implementation of View-3. A reader interested in the details is referred to a more complete description [Kratzer84].

II.1 The Basic Objects in View-3

There are three kinds of objects in View-3: distributed processes, processes and ports. A distributed process is intended to model a single processor and consists of a set of processes. Each process is a schedulable entity.

Processes communicate with each other through ports. The port mechanism of View-3 supports buffered, byte-stream channels. A port may be thought of as a loose wire which can be attached to a pair of processes. When the ports (wires) are connected to both processes, the processes may communicate with each other. Connections of process ports may be specified at the time the processes are created. Alternatively, a port may be created, connected or disconnected dynamically. Besides providing a conduit for data transfer, the port can add time-of-day information to data as they move from one process to another. Additionally, I/O devices and files also communicate with processes through ports.

II.2 Description of Objects

Each kind of object is described by a template. These templates are used by View-3 to activate processes and control communication between them. The templates contain all the information needed for a process's creation. In the current implementation, this includes:

Process Name	How the process should be identified by the View-3 system.
Executable File	Where a runable version of the process code can found.
Port Templates	A list of templates to be used for creating the ports owned by this process.

A template for a distributed process consists of a set of process templates and a set of port templates; a template for a port consists of port parameters: whether messages are to be time stamped, and if the port should be connected to a file, an I/O device or another port.

Proceedings of the Washington Ada Symposium '85;
March 25-27
pages 121-127

The user has two ways to describe templates for distributed processes: a language-based system and a graphics-based description system. The language is a typical configuration description system where one declares the distributed processes, processes and ports, and how these objects are to be connected. The system interactively queries the user for parameters and builds the templates from the responses.

The graphics-based description system allows one to draw a picture of the process structure that a distributed process should have. The process and port templates are derived from the graphical representation. This picture is created by the programmer by selecting various kinds of boxes that the system provides. (In this respect, View-3 is similar to ThingLab[Borning81]. However, unlike ThingLab, View-3 provides a finite number of object types and applies predefined interpretations for them.) The definition of a distributed proces containing a reader process and a writer process is shown below [2].

```
+-----------------------------------------------+
| Dist. Proc:  |                                |
|              |                                |
|    RnWs      |                                |
|              |                                |
|--------------|          ----------            |
|              |          | Reader |            |
| >Add process |          ----------            |
|  "Reader"    |              |                 |
|              |              |                 |
| >Place       |              |                 |
|  process     |          ----------            |
|  "Reader"    |          | Writer |            |
|              |          ----------            |
|              |                                |
+-----------------------------------------------+
```

The positioning of the processes and their ports is done by placing the cursor at the place desired. As implemented on DEC[1] Gigi terminals, this positioning is done by using the arrow keys. We expect that future versions of the system would use a better pointing device and display system, such as a mouse and a bit-mapped display. We feel that the interface provided for the Blit Debugger [Cargill83] is more appropriate for both template design and process monitoring (discussed in the next section).

II.3 Manipulation of the Objects

Using these templates, the View-3 system implements tools that enable a programmer to start a set of communicating processes and to monitor their execution on the terminal. The run-time display is graphics oriented, allowing the user to monitor any process that is running. Several display formats are provided. One of the most descriptive formats consists of three windows: a window for communicating with the View-3 system, and two windows for showing communication between two processes. A typical display including windows for two processes (called StartUp and Clock) and the View-3 system window is shown below[2].

In this illustration, communication between processes is depicted in the two windows labelled Process Interactions. Messages leaving a process are designated by the symbol "->"; received messages are indicated by the "<-" symbol. Each line in a process interaction window gives the message's time stamp and the (process) name of the sender or receiver of the message. Using this format, the View-3 screen formatter aligns messages in the two process interaction windows based on message time stamps. This allows the programmer to compare message traffic.

There are several important points illustrated by this example: First, the system allows the user to communicate with processes and View-3 at the same time. Second, View-3 allows multiple processes to share a single terminal. In a system of distributed processes, each set of processes may operate under the assumption that each has its own terminal. Therefore, there must be a way for a collection of processes to share a single terminal. In View-3, process communication is placed in separate windows which are tagged with the process's name. Third, synchronization of processes is easily visible because time stamps are used to align messages. Because the sender and receiver are identified, the programmer can easily see how messages move between processes. Alignment of messages based on time stamp very graphically shows when certain events were synchronized. Because control of synchronization is an important aspect of systems with interacting processes, a tool should provide a clear representation of the synchronization between processes. This display, running in real time, provides this kind of aid.

```
   Process StartUp        Process Clock
+-----------------------------------------------+
|StartUp:               |Clock:                 | <-- Process Interactions
|                       |                       |
|1229: "Now" -> Clock   |1229: "Now" <- StartUp |
|                       |1230: "1230" -> Synch  |
|1245: "XYZ" <- Queue   |1245: "1245" -> Synch  |
|                       |1300: "1300" -> Synch  |
|1300: "1300" <- Clock  |1300: "1300" -> StartUp|
|                       |                       |
|-----------------------------------------------|
|>                                              | <-- View-3 Command Window
|>                                              |
+-----------------------------------------------+
```

II.4 Implementation of View-3

This implementation of View-3 runs on a Vax/UNIX[1] system (running Berkeley Version 4.1), is written in C and is intended to support processes written in C. The implementation consists of a two level hierarchy of function. The lower level of function, the View-3 kernel, implements the View-3 objects and the operations on them. The kernel consists of a set of subroutines that reside in the View-3 library and that are linked into View-3 process images at link-time. The upper level of function consists of a set of View-3 processes that create the user interface to View-3. The basic structure of the upper level is shown below.

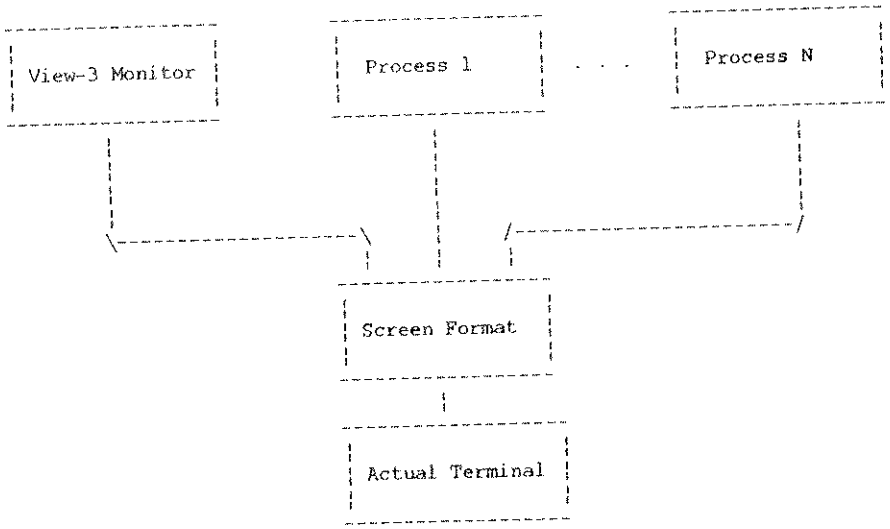
A process, written in C, may use the kernel supported operations on View-3 objects. As such, the kernel is responsible for carrying out requests from processes for port operations (creations, destructions, connections and disconnections,) and for operations on distributed processes and processes (creations, destructions, and synchronization.) The View-3 Monitor allows the user to start and stop the execution of distributed processes and processes; it also provides status information on what is running. In creating a distributed process or process, the Monitor collects information on how processes are wired together and passes this information to the Screen Format module which controls the display format of the terminal.

III. Experience with View-3

Our experience with View-3 is admittedly limited. Since we are currently building up several research projects that use distributed programs, we feel that we must have tools to assist our efforts. View-3 is an appropriate tool for a number of reasons. The graphical nature of interaction with the View-3 system is important. We can easily create, using the template definition system, a picture of how the different pieces of a distributed system are connected. When a distributed program is run, the Monitor can display the actual topology of the distributed process during execution; therefore, changes in the topology, which are the result of explicit process destroys or creates as requested by the programmer, are readily apparent. The windowing display formats provided by Screen Format clearly show actual process interaction, making deviations from intended behavior obvious and clear. Because people can more easily see differences in a pictorial representation more readily than small differences in print-outs of variable contents, the View-3 system provides a simple way to see if the program is working correctly. Indeed, our experience seems to indicate that a tool that can clearly show the interaction of processes without providing the exact details of the interaction is useful for finding synchronization errors. However, once an error is detected, detailed examination of the contents of variables or messages is usually necessary for locating the exact source of the error.

As mentioned before, the View-3 system was built on top of Berkeley UNIX 4.1 running on a Vax 11/750 and was designed to use Gigi terminals. The system was built as a "guest" or "client" layer. The underlying UNIX mechanisms for process scheduling and for interprocess communication were used. This strategy for implemenation significantly affected the performance and ease of use of the system.

The performance of the View-3 system is slow, largely due to the overhead associated with UNIX processes. We decided to use UNIX processes rather than writing our own process scheduler for three reasons: First, because of limited manpower, we wanted to do as little additional work as possible. Second, because we expect later versions of View-3 to be physically distributed, we wanted the process interface to be the same regardless of whether the distributed process was local or remote. Third, because we were constrained to use university-wide, nonresearch facilities, we could not change the UNIX kernel to provide better process and scheduler support for View-3.



In contrast, the implementation of an APSE tool could overcome all of the limitations noted above and could offer good performance. Normally, an APSE tool like View-3 could directly use the task scheduler of the Ada run-time system, which in turn we would insist be efficiently implemented in the kernel level of the target machine. Further, we could guarantee the uniform interface for distributed and local tasks by the very fact that all tasks communicate using the interface provided by the rendezvous mechanism.

Although the execution speed is not as good as we would like, the use of graphics to define templates and to run a distributed system has been a valuable aid. When building distributed systems, one generally does not view the system in terms of specific accept statements but rather in terms of protocols, message flows or synchronization patterns. View-3 provides a very effective graphic representation of these kinds of process interactions.

From our experience and others [Bates83], we conclude that tools should permit one to express process interactions in as a high level as possible and then the tool should generate the necessary information for the actual connection of processes in the described way. When the processes are running, it is equally useful to be able to view the original model and the actual run-time structure in a convenient way to validate that the written system in fact performs as desired.

IV. Integration of View-3 into APSE

The original design of View-3 was intended to provide a useful debugging aid for students who were building multiprocess systems. As such, it is only one of many tools that we would expect a programming team to have access to during system development. A preliminary definition of such a tool box is provided by the KAPSE/MAPSE/APSE documents, for example [Buxton80]. In looking at these documents, one sees requirements for standard debuggers, loaders, binders, performance monitors and configuration controllers. The proposed tool boxes do not contain any specifications for an interactive task-debugging tool. Therefore, the procurement of a tool like View-3 must be carefully specified in addition to a standard KAPSE system. Such specifications must take into account the different approach used by a a View-3 tool as compared to a conventional tool. Some of these differences and implications are discussed below.

IV.1 Monitoring vs Stopping a System

Most of the tools currently available support execution monitoring of a single process that can be stopped or started at will. Perhaps the most prevalent such tool is the symbolic debugger. Such a tool can only be effectively used when the interactions of a system can be captured in a single snapshot. From any snapshot, one can slowly move the program forward (and in some systems, backwards [Brown82]) and observe how the system behaves. Logically extending this technique, a snapshot of a system with multiple processes would list out the state of each process whenever the system was stopped. However, stopping a system also adds another point of synchronization in the system, namely at the place where the system was stopped.

Although such a stop-and-examine technique can provide valuable information about a program, we believe that a more dynamic view of the process interactions is required. A tool like View-3 is different from most tools in that a programmer is watching a running system rather than snapshots. Therefore, the information provided to the programmer must be gleaned, in part, from the system tables maintained by the run-time system. To provide access to this information may require support from the compiler, run-time system and terminal handler. Thus, a requirement for a View-3 like tool should be proposed early, since it will probably be hard to retrofit into an existing Ada implementation[3].

If a stop-and-examine technique is included in a tool, we feel that large scale manipulation of processes should be allowed in addition to stopping and starting processes that are already underway. The View-3 system shows how these capabilities can be provided in an easy to use system.

IV.2 Models of Process Interaction

As it stands, the source for View-3 cannot be transformed into Ada and added to a running APSE. The aids provided by View-3 are very closely tied to a particular model of process interaction, which is derived from UNIX, that has similarities and differences with the Ada rendezvous model. Differences include: View-3 transports typeless information between ports; in Ada, all information passed to an accept statement is typed. View-3 uses asynchronous message passing; Ada uses a synchronous rendezvous mechanism. View-3 assumes that it can control the run-time environment of the executing processes; an APSE tool (as opposed to a KAPSE tool) must be a "guest" or "client" program that can run on top of an already existing KAPSE and not replace part of the KAPSE. View-3 presumes the ability to dynamically define and create processes; Ada allows the creation of those only processes that can be named within the scope of the creator. Since an Ada implementation of View-3 would exist in an outer (or even disjoint) scope from the processes being manipulated, there is no easy way to dynamically create the processes being tested from within View-3 systems. None of these problems is insurmountable. However, some require more effort than offers. We briefly discuss these considerations.

The exchange of typed information rather than typeless presents some difficulty in displaying information from ports. The screen manager must be able to decode messages of arbitrary types and to present them on the screen. Part of the problem may be solvable if the View-3 screen manager has access to the same kind of information used by the symbolic debugger for presenting the static state of the program. Presumably, a type descriptor can be used for interpreting bits as a particular type.

The use of type descriptors has the disadvantage of connecting the View-3 tool to the particular implementation of Ada that exists in the APSE. A implementation independent approach may require the use of some auxiliary functions to translate between arbitrary abstract types and some predefined types that View-3 would understand. This approach has been used elsewhere for writing data abstractions that could be transmitted over a network [Herlihy80, Herlihy82]. If such auxiliary functions were automatically generated by the template creation system, then the resulting View-3 could be transported from one Ada implementation to another transparent to the programmer.

The use of synchronous rendezvous mechanism in Ada vs an asynchronous message passing mechanism in View-3 is actually a minor restriction. What is important to note is that the aid should provide several models of process abstraction which can be translated into Ada tasks. Such translations for path expressions into Ada already exist. A more general method for creating a diverse set of control structures has also been proposed [Penedo81]. Adding message passing and monitor schemata should not be insurmountable. Therefore, one should not think of any aid as forcing the programmer into one particular process model, but rather as providing the programmer with a set of models that can be translated automatically by the tool into Ada tasks.

The need for a View-3 environment to interact with the KAPSE can be accomodated if the MAPSE does not hide the entire facilities of the KAPSE. In short, the KAPSE need only provide the entry points normally used by the translation system for its manipulation of tasks. In an ideal situation, the task scheduler could be changed to a user-provided scheduler, much in the same way that some systems allow guest programs to add new device drivers to the device tables maintained by the kernel [Dartmouth83]. We assume that such a mechanism would also allow the dynamic creation and deletion of tasks.

The scoping and naming problem in Ada is more serious. An example of this problem arises when trying to write a task scheduler that can work with any task type (It results because Ada is strongly typed.) For a task scheduler, some programmers feel that each task should be assigned a unique integer when the task is defined in advance of run-time scheduling. Such a static assignment of task id's is not acceptable since it requires the programmer to alter the source program to accomodate the monitoring system. We feel that a reasonable approach may rely on a generic monitor package that will contain some procedures, written in a language more weakly typed than Ada, such as C, which can manipulate differently typed tasks. Another approach would require the Ada source for the user's processes (tasks) and the monitoring system be generated by the template generator. With a properly open KAPSE and with a template generator that directly produces Ada code to be used by the programmer, it is reasonable to provide a View-3 like system in the APSE.

V. Conclusions

The purpose of this article is provide a manager who wishes to procure a task-debugging tool with an example of one such tool and how this tool might be used by a project. Therefore, we feel that it is fitting to conclude with a checklist that might be used when specifying such tools:

1. There should be graphically oriented interface that permits the programmer to specify how Ada tasks should interact. With such a specification, the debugging aid should then produce Ada code which can organize a system for execution. This code should contain pragmas that can be used by the translation system to avoid the overheads of a general rendezvous and produce far more efficient code, e.g. pragma(monitor). (This idea is more fully discussed elsewhere [Hilfinger81].)

2. There should be a way to graphically present the state of more than one task simultaneously. With such a display, it should be easy to tell when a rendezvous has occurred, when a task is blocked waiting for an accept statement and when a time-out has occurred.

3. When debugging a system of tasks, the programmer should be able to create new tasks from previous defined task types and to insert them into the running system. The tool should keep a record of such interactions so that corresponding changes to the Ada program can be made manually or automatically (such as the Paddle system [Wile82]).

4. The programmer should be able to terminate tasks that are currently running. This feature allows the programmer to test the robustness of software slated for failure-prone hardware in embedded systems.

5. The debugging tool should come with a set of process models that can be used by Ada tasks. As a minimum, we suggest that a simple word oriented unbuffered message passing facility be provided, though a buffered facility and a facility that could transmit abstract objects would be very useful. Models using protocols, path expressions, guarded commands, polling, cyclic schedulers and monitors should eventually be provided.

The suggestions above pertain only to the functionality of the proposed tool. We also believe that the certain facilities need to be present in the APSE for this and other kinds of task debugging tools:

1. The APSE must provide access to the task scheduler. This is needed to allow task debugging aids to alter the circumstances under which a certain task runs. In the case of View-3, there must be a way to guarantee that the Ada task which controls the View-3 system receives priority for its execution.

2. The APSE must provide access to the task creation and destruction mechanisms. Tools like View-3 need some way to link in new processes without writing Ada source code, compiling it and then dynamically linking it in. One way is to provide some skeleton object code that can then be added to the scheduler's process list as required.

3. The APSE must provide access to the rendezvous mechanism. As mentioned in the notes on implementation, the UNIX interprocess facility was too slow for use by client programs to implement efficient interprocess monitoring. This same fate probably awaits task-oriented debugging tools that cannot directly access the tables and library routines needed to control rendezvous.

Work on these kinds of tools is still in its infancy, and so we do not expect production quality aids overnight. However, the impetus for building such aids can come from managers who learn of the value of these tools and then start requiring some kind of similar support for their next project.

VI. References

[Bates83]

Peter Bates and Jack C. Wileden, "An Approach to High-Level Debugging of Distributed Systems," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Sigplan Notices, Vol. 18, No. 8, August 1983, pages 107-111.

[Buxton80]

J. N. Buxton, "Requirements for Ada Programming Support Environments," ("Stoneman"), United States Department of Defense, February 1980.

[Borning81]

Alan Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," ACM Transactions on Programming Languages and Systems, 4(3):353-387, October, 1981.

[Brown82]

Marc H. Brown and Steven P. Reiss, "First Steps Toward a Computer Science Environment for Powerful Personal Machines," Technical Report CS 83-04, Department of Computer Science, Brown University, Providence, RI 02912, December 1982.

[Cargill83]

Thomas A. Cargill, "The Blit Debugger," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, Sigplan Notices, Vol. 18, No. 8, August 1983, pages 190-200.

[Dartmouth83]

Dartmouth College, Keiwit Computation Center, DCTS Documentation, "TM059: System Programmer's Reference Manual," November 1983.

[Herlihy80]

Maurice Peter Herlihy, "Transmitting Abstract Values in Messages," Technical Report MIT/LCS/TR-234, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. 02139, April 1980.

[Herlihy82]

M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types," ACM Transactions on Programming Languages and Systems, 4(4):527-551, October 1982.

[Hilfinger81]

Paul N. Hilfinger, "Abstraction Mechanisms and Language Design," Technical Report CMU-CS-81-147, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. 15213, June 1981.

[Houghton83]

Raymond C. Houghton, Jr., "A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE)," Ada Letters, Vol. III, No. 3, November/December 1983, p. III-3.69-III-3.78.

[Kratzer84]

Ann Kratzer, "The View-3 Process Monitoring System," Technical Report COSC-84-01, Dartmouth College, Department of Mathematics and Computer Science, Hanover, NH, 03755, January 1984.

[Penedo81]

Maria Heloisa Penedo, Daniel M. Berry and Gerald Estrin, "An Algorithm to Support Code-Skeleton Generation for Concurrent Systems," Proceedings of the 5th International Conference on Software Engineering, IEEE Catalog No. 81CH1627-9, March 9-12, 1981, pages 125-135.

[Wile82]

David S. Wile, "Program Developments: Formal Explanations of Implementations," Research Report ISI/RR-82-99, Information Sciences Institute, Marina del Rey, CA. 90291, August 1982.

VII. Notes

[1] Ada is a trademark of the United States Government. DEC and VAX are trademarks of the Digital Equipment Corporation. UNIX is a trademark of Bell Laboratories.

[2] The actual displays contain more information than is illustrated in this diagram. Further, smoother graphics are used instead of the line printer dashes and hyphens.

[3] Because our system was built on top of C, which has no primitive process facilities, we simultaneously developed the process model, scheduler and View-3. Had this information been part of the C compiler instead, we might not have been able to produce View-3 without writing parts of the compiler.