Faculty and Researcher Publications | Faculty and Researcher Publications

1992-01

Computer-Aided Prototyping for a Command-and-Control System Using Caps

Luqi

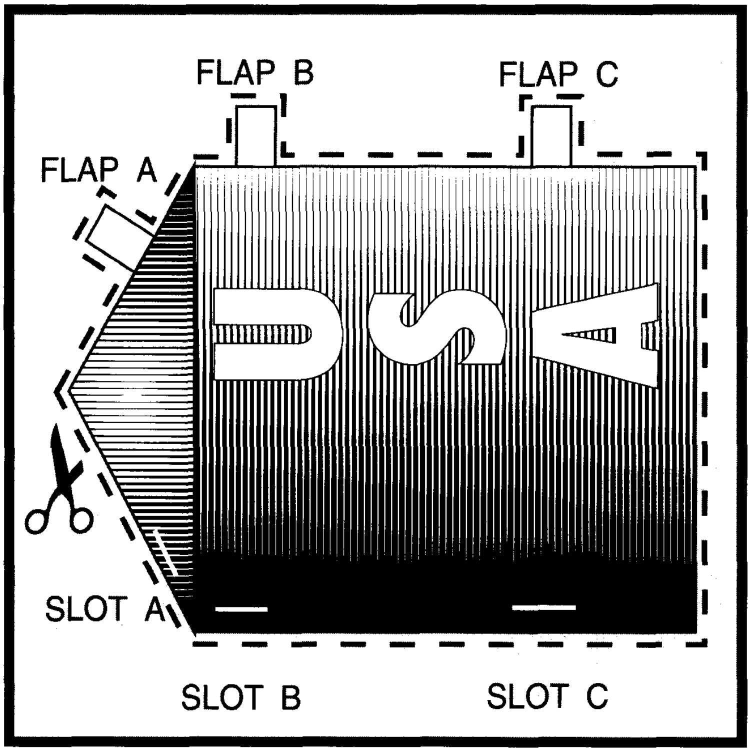
IEEE

http://hdl.handle.net/10945/43625

COMPUTER-AIDED PROTOTYPING FOR A COMMAND-AND-CONTROL SYSTEM USING CAPS

This case study shows the feasibility of using computer-aided prototyping to validate a C^3I system's requirements and describes the enabling technology.

LUQI
Naval Postgraduate School



Computer-aided prototyping, which seeks to automate early design phases, is an important technique for developing complex embedded systems that have strict time constraints. System analysts and users need prototyping methods to adequately formulate and assess the requirements for those systems. They can then use computers to apply these methods rapidly.

At the Naval Postgraduate School, my colleagues and I have recently completed an experiment to evaluate our rapid-prototyping methods and computer-aided design environment.

Our experiment was to prototype a generic command, control, communications, and intelligence station[1] and generate the Ada code from the prototype's specifications automatically. The results show that it is feasible to use computer-aided prototyping for practical, real-time Ada applications.

C^3I applications are difficult to develop, for the reasons outlined in the box on p. 58. The C^3I prototype we developed had characteristics typical of embedded software, including distributed processing; hard real-time constraints; multiple, predefined hardware interfaces; and com-

0740-7459/92/0100/0056/$03.00 © IEEE

plex requirements. We generated a color, multiwindow executable Ada prototype that can process tactical data from multiple interfaces in real time.

We used the prototype to get feedback about the proposed design's effectiveness, performance, and structure and to evaluate the soundness of our design decisions. The feedback helped us improve and refine requirements and evaluate the feasibility of the functional specification. We iteratively refined and validated requirements by modifying an operational prototype until users were satisfied with its behavior.

We used the Prototype System Description Language[2] and Computer-Aided Prototyping System[3] in our experiment. PSDL integrates the tools in CAPS, which help the designer create the design, automatically construct a real-time schedule, and automatically generate an executable Ada model of the proposed system from the PSDL specification. The Ada model is a combination of CAPS-generated Ada programs and reusable atomic Ada components.

CAPS also supports system management and helps control a system's evolution.[4] This support helps designers give timely responses to modification requests and helps protect the system's integrity as it evolves, extending its life.

SYSTEM REQUIREMENTS

A C^3I system helps military officers understand tactical situations: It provides communication among officers on different platforms and external forces, and it processes tactical data from various internal and external sources, such as radar and sonar.

Structure. The proposed C^3I system is a network of generic C^3I stations, each of which is a specialized instance of a common design. The network is a large, geographically distributed system that may have many thousand nodes. Each station is mounted on a platform whose location typically is not fixed. Larger platforms can have several stations serving officers with different responsibilities.

Each station can be viewed as a single embedded system or a local distributed system with multiple processors. A subgoal of our research is to establish the feasibility of a low-cost C^3I system consisting of a loosely coupled network of C^3I stations installed in sites without substantial C^3I support, like noncombatant ships or small combatant platforms.

Each station would be a generic C^3I station, although individual configurations could provide tailored subsets of functionality. Steve Anderson's report gives a detailed description of the generic C^3I station's requirements.[5]

Interfaces. Figure 1 shows a single-user C^3I station and its external interfaces to the user and to the weapon systems, platform sensors, navigation system, and communication links. The information the user requires includes the platform's location, the status of its weapon systems, and the locations and characteristics of other platforms in the area. The station receives and transmits track information and command-and-control data via communication links, receives track information from platform sensors, outputs a tactical display to the user, provides a text editor for generating and sending messages, and provides a way to verify and maintain track-data integrity. (A track is the system's representation of an external object such as a platform or a navigation hazard. Tracks contain information about the location and characteristics of the external object.)

The user is an officer at some command level. The station is the officer's communication channel to superiors, subordinates, and other officers at the same command level. The user communicates with the station via a keyboard, graphical display, and pointing device to obtain information about selected tracks, the status of the host platform and C^3I system, and messages from other officers. The user may update track information, control the status of the C^3I system, and originate messages.

The antennas, notch filters, and data-terminal sets provide communication

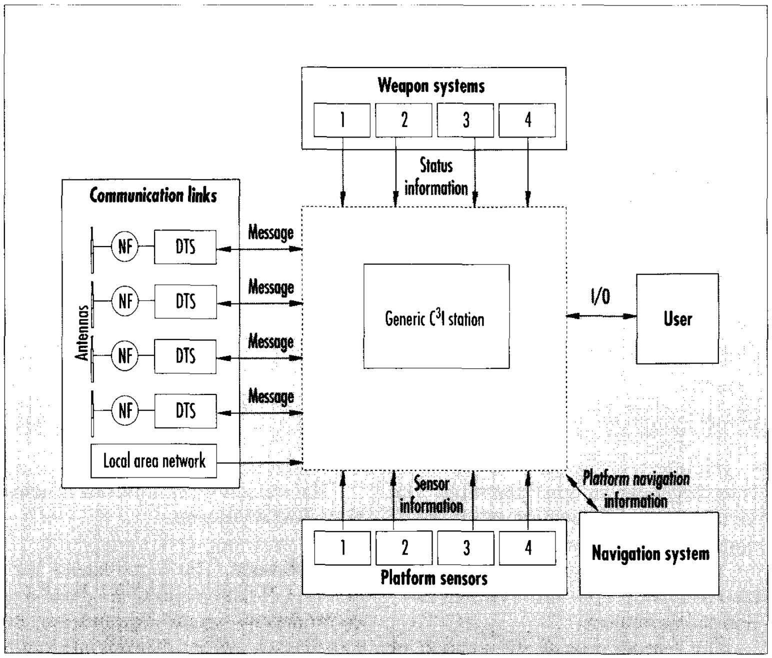


Figure 1. Diagram of single-user generic C^3I system and its external interfaces.

NATURE OF C^{3}I SYSTEMS

C^{3}I systems help military officers understand tactical situations. They are difficult to develop because

- ♦ Their use in strategic defense applications makes correctness and reliability critical.
- ♦ They are influenced by many people, by organizations, and by policies, so their requirements are complex and difficult to determine.
- ♦ Their design depends on techniques to guarantee that hard real-time constraints will be met both in large distributed systems connected by long-haul networks and in local distributed systems with many hardware structures. Current software research has not solved many of these systems' problems, like real-time-database design, network-flow prediction, upper bounds for the actions of real-time operating systems, hard real-time algorithms for general problems, and robust identification of processes in distributed systems.
- ♦ Their complex, dynamic interfaces make it almost impossible to deal with changes in requirements.
- ♦ As with any large system, their development is costly, and the current low productivity of software development aggravates the problem.

We use prototyping and computer-aided design techniques to address many of these difficulties.

TABLE 1
DIALOGUE-RESPONSE TIMES

Type	Response time
Question and answer	0.5 to < 2.0 seconds
Menu selection	< 0.2 second
Form filling	> 2.0 seconds
Function keys	< 0.2 second
Command language	0.5 to > 2.0 seconds
Natural/Query language	0.2 to < 0.5 second
Graphical interaction	< 0.2 second

TABLE 2
MESSAGE-DELAY TIMES

Message precedence	Time between message completion and transmission	Time between message reception and display
Flash	< 1 second	< 1 second
Immediate	< 2 seconds	< 2 seconds
Priority	< 3 seconds	< 3 seconds
Routine	< 4 seconds	< 4 seconds

links to stations on other platforms. The local area network connects to other stations on the same platform, if any.

The navigation system provides information about the platform's current location and movement.

The sensors provide the location of surrounding platforms.

The weapon systems provide information about their status.

Requirements. The requirements for a generic C^{3}I station include hard real-time constraints on system responses. Any design for such a station depends on assumptions about the timing characteristics of the external systems with which it interacts. Because accurate values for many of the hard real-time constraints in a C^{3}I system are classified, we based the design of our unclassified prototype on seven arbitrary assumptions:

- ♦ It should be able to retrieve up to 1,000 tracks in less than one second.
- ♦ It should enter the contents of a track-data message into a track database in less than two seconds.
- ♦ It should conform to the dialogue-response and message-delay times summarized in Tables 1 and 2.
- ♦ It has four sensors, four weapon systems, and four communication links.
- ♦ Its navigation system updates velocity every 41 ms, transmits velocity every 983 ms, and updates latitude and longitude every 1.3 seconds.
- ♦ Its platform sensors track a maximum of 100 tracks per sensor per second.
- ♦ Its weapon systems update their status once every second.

We did not consider network delay because the focus of this requirements analysis and prototyping effort was on timing constraints within individual stations.

PROTOTYPE SLICE

The prototype includes a generic C^{3}I station and its interacting external systems. We formulated the prototype as a closed system because we must simulate the external systems to demonstrate the proposed behavior of the C^{3}I station. (Vedat Coskun and Cengiz Kesoglu[6] provide complete details of the prototype.)

Figure 2 shows a representative slice of the PSDL definition that contains a part of the system related to message routing (see the box on p. 62-63 for an overview of PSDL). The slice takes a path from the hierarchically structured prototype's root to its leaves. The root is a single PSDL operator, c3i_system, which is decomposed into more primitive operators. The designer defines the decomposition via a PSDL graph like that shown in Figure 2.

Timing requirements. Figure 2 defines the control constraints for the operators in its graph. Each operator definition includes its timing constraints, based on the requirements outlined earlier. For example, the minimum calling period for the sensor_interface operator is 2,500 µs, which was derived from our assumption that the maximum data rate from each of the four sensors is 100 tracks per second, so the minimum calling period is one second divided by 400, or 2,500 µs. This is the longest time the system can allow between consecutive firings of the sensor_interface operator in the static schedule.

The requirements state that the maximum delay between receiving a track message and entering it into the database is two seconds. In the initial design of the prototype we allocate this delay evenly between the sensor_interface and the track_database_manager operators, leading to maximum response times of one second each. We may reallocate these constraints later as we explore requirements in more detail.

As Figure 2 shows, we don't specify the timing requirements for the communication interface at this level because they are influenced by two separate requirements: the maximum delay of a communication message (one second), and the maximum delay of a track message (two seconds). Because each requirement is likely to affect components on different dataflow paths, we define the corresponding timing constraints at the next decomposition level.

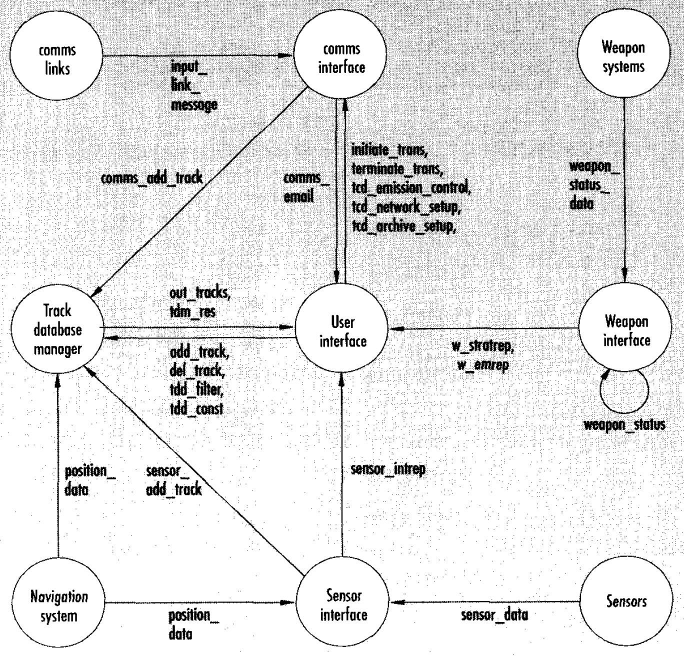
In the initial prototype version, we do not distinguish timing requirements for different message classes or different types of user interaction. Instead, we design for the worst case: All messages must be delivered within one second and all user-interface functions must complete within 200 ms. We may relax these assumptions in later iterations if we find it is not feasible to meet these simplified requirements. At present, we introduce distinctions only to show the feasibility of the timing requirements.

The BY REQUIREMENTS clauses in Figure 2 document by keyword the requirements from which we derived the timing constraints.

Communication interface. The communication interface performs functions di-

```
OPERATOR c3i_system
  SPECIFICATION
    DESCRIPTION {This operator represents the entire prototype.}
  END

IMPLEMENTATION GRAPH
```

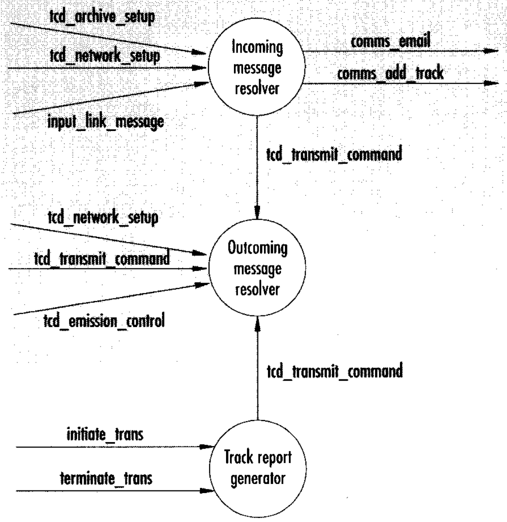


```
DATA STREAMS
– Type declarations for the data streams in the graph go here.

CONTROL CONSTRAINTS
  OPERATOR comms_interface
  OPERATOR sensor_interface
    MINIMUM CALLING PERIOD 2500 microsec BY REQUIREMENTS sensor_rate
    MAXIMUM RESPONSE TIME 1 sec          BY REQUIREMENTS track_delay
  OPERATOR track_database_manager
    MINIMUM CALLING PERIOD 1 ms
      BY REQUIREMENTS track_retrieval_rate
    MAXIMUM RESPONSE TIME 1 sec
      BY REQUIREMENTS track_delay
  OPERATOR user_interface
    MAXIMUM RESPONSE TIME 200 ms
      BY REQUIREMENTS dialogue_response_time
  OPERATOR weapons_interface
    TRIGGERED BY SOME weapon_status_data
    OUTPUT weapons_emrep IF
      weapon_status_data.status = damaged OR
      weapon_status_data.status = service_required OR
      weapon_status_data.status = out_of_ammunition
    MINIMUM CALLING PERIOD 250 ms
      BY REQUIREMENTS weapons_status_rate
    MAXIMUM RESPONSE TIME 1 sec
      BY REQUIREMENTS weapons_status_rate
  OPERATOR comms_links
  OPERATOR sensors
  OPERATOR navigation_system
    PERIOD 1300 ms BY REQUIREMENTS navigation_rate
  OPERATOR WEAPONS_SYSTEMS
    PERIOD 250 ms  BY REQUIREMENTS weapons_status_rate
END
```

Figure 2. Slice of PSDL definition for message routing, including a decomposition graph.

```
OPERATOR comms_interface
SPECIFICATION
 INPUT input_link_message
        tcd_transmit_command   :transmit_command,
        tcd_emission_control   :emissions_control_command,
        tcd_network_setup      :network_setup,
        tdd_archive_setup      :archive_setup,
        initiate_trans         :initiate_transmission_sequence,
        terminate_trans        :boolean
 OUTPUT comms_email            :filename,
        comms_add_track        :add_track_tuple
DESCRIPTION {This operator is responsible for processing incoming and
             outgoing messages as well as producing track reports and
             converting message formats }
END
IMPLEMENTATION GRAPH
```



```
CONTROL CONSTRAINTS
  OPERATOR incoming_message_resolver
    MINIMUM CALLING PERIOD 40 ms
      BY REQUIREMENTS link_speed
  OPERATOR outgoing_message_resolver
    MINIMUM CALLING PERIOD 71 ms
      BY REQUIREMENTS transmission_rate
    MAXIMUM RESPONSE TIME 800 ms
      BY REQUIREMENTS message_delay, dialogue_response_time
  OPERATOR track_report_generator
    PERIOD 2 sec
      BY REQUIREMENTS track_reporting_rate
    TRIGGERED IF not terminate_trans
END
```

Figure 3. *PSDL definition for the communication interface.*

rectly related to message reception and transmission. Because stations use different communication equipment, this module's implementation will vary greatly from one instantiation to another. However, all generic C^{3}I stations are subject to a common set of behavioral and timing requirements. The requirements dictated that the implementation of this interface be very modular, to isolate site dependencies.[5]

Figure 3 shows the PSDL definition for the communication interface. This interface must monitor, relay, and transmit messages on various networks within hard real-time deadlines. It filters, routes, sorts, and translates messages, and it analyzes messages arriving at the communication interface to determine if they contain track information, if they must be relayed to other participants in the network, and if they must be archived.

Constraints. At a network speed of 1,300 to 5,000 bps and assuming the shortest message is about 100 characters (800 bits, minus start/stop bits and redundant data), the system will receive a maximum of 25 messages from four links in one second. Therefore, the minimum calling period of the incoming_message_resolver is one second divided by 25, or 40 ms. We leave the specification of the maximum response time to the next level because this operator processes both messages containing tracks (comms_add_track) and messages containing communications among officers (comms_email).

We estimate the maximum number of outgoing messages in one second as

- one message every two seconds from the track_report_generator (a periodic operator) plus
- 12.5 messages every second to relay (assuming that half the messages received should be relayed) plus
- one message every second from the user (an assumption)

for a maximum of 14 messages per second. Thus, the minimum calling period for the outgoing_message_resolver operator is one second divided by 14, or 71 ms. Its maximum response time is one second minus 200 ms, or 800 ms, since we assume messages must be transmitted within one second of completion, and that the user interface can have up to a 200-ms delay.

The requirements give no time constraint for track reports. The prototype is designed to produce track reports every two seconds when this feature is activated, based on the analyst's assumption of a reasonable reporting rate. This assumption must be validated and adjusted as necessary.

Incoming messages. The incoming_message_resolver is still too complicated for direct implementation, so it is decomposed into four atomic operators, shown in Figure 4.

The maximum time delay for a message is one second. The user interface requires 200 ms, and the remaining 800 ms are initially allocated evenly between the input_file_parser and the message_type_decision.

The time delay for a track message can be at most two seconds. The track database manager requires one second, and the input_file_parser and message_type_decision have been allocated 400 ms each, leaving 200 ms for track_extractor.

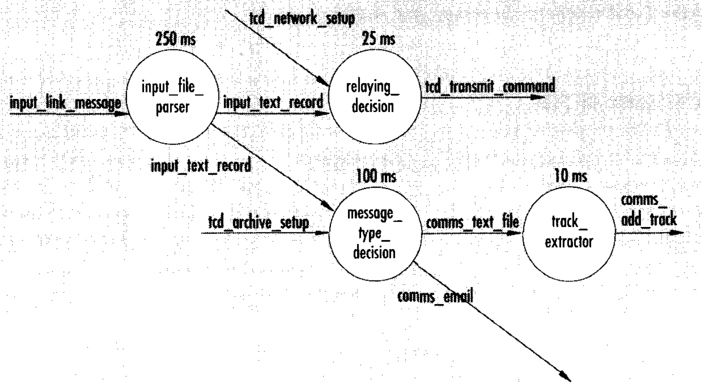
The maximum response time allowed for relaying_decision depends on the maximum delay allowed between the receipt and transmission of a relayed message. The requirements do not contain this information, so the analyst makes an arbitrary assumption that this delay should be no more than 1.5 seconds. Because the outgoing_message_resolver takes 800 ms and the input_file_parser takes 400 ms, this leaves 300 ms for relaying_decision.

The analyst annotates each atomic component in the graph in Figure 4 with these execution-time estimates. The CAPS tools use these annotations to find reusable components from the software base.

Triggering conditions are designated as BY ALL because they must execute for every incoming data value. This implies input_text_record and comms_text_ file must be dataflow streams, and the execution rates of the operators input_file_parser, message_type_decision, track_extractor, and relaying_decision must be synchronized. The output guards on message_type_decision suppress output when the incoming message is neither email nor track information.

To complete this top-down slice, Figure 5 shows the PSDL specification for one atomic operator in the prototype, message_type_decision, which decides if the input_text_record contains track information. The CAPS graphical editor derives this operator's maximum execution time and its I/O interface mechanically from the implementation graph of its parent operator. The specification serves as the basis for the retrieval of a reusable component or for a manual implementation effort. As indicated in the PSDL description, this operator is implemented in Ada.

```
OPERATOR incoming_message_resolver
SPECIFICATION
  INPUT    input_link_message:filename,                   :filename,
           tcd_network_setup:network_setup,               :network_setup,
           tdd_archive_setup:archive_setup                :archive_setup
  OUTPUT   comms_email:filename,                          :filename,
           comms_add_track:add_track_tuple,               :add_track_tuple,
           tcd_transmit_command:transmit_command          :transmit_command
  DESCRIPTION { This operator processes the incoming communications
                messages }
END
IMPLEMENTATION GRAPH
```



```
  DATA STREAMS
    input_text_record,comms_text_file : text_record
  CONTROL CONSTRAINTS
    OPERATOR input_file_parser
      MAXIMUM RESPONSE TIME 400 ms
      TRIGGERED BY ALL input_link_message
    OPERATOR relaying_decision
      MAXIMUM RESPONSE TIME 300 ms
        BY REQUIREMENTS relaying_delay
      TRIGGERED BY ALL input_text_record
      OUTPUT tcd_transmit_command IF
               input_text_record.relayed
    OPERATOR message_type_decision
      MAXIMUM RESPONSE TIME 400 ms
      TRIGGERED BY ALL input_text_record
      OUTPUT  comms_text_file IF comms_text_file.is_track
      OUTPUT  comms_email IF NOT comms_text_file.is_track
    OPERATOR track_extractor
      MAXIMUM RESPONSE TIME 200 ms
      TRIGGERED IF comms_text_file.is_track
  END
```

Figure 4. *Four atomic operators for incoming_message_resolver.*

```
OPERATOR message_type_decision
SPECIFICATION
INPUTinput_text_record : text_record,
tdd_archive_setup : archive_setup
OUTPUTcomms_text_file : text_record,
comms_email : filename
MAXIMUM EXECUTION TIME 100 ms
DESCRIPTION { Sets the is_track field of comms_text_file if
 input_text_record contains track information. }
END
IMPLEMENTATION Ada message_type_decision
END
```

***Figure 5.** PSDL definition for message_type_decision.*

USING CAPS

To generate the prototype's Ada code, we used CAPS and the Transportable Applications Environment Plus, a windowing package developed at the National Aeronautics and Space Administration's Goddard Space Flight Center.[7] TAE Plus provides either Ada or C code to create the user-interface modules, but we had to modify the generated code to make it fit the CAPS coding conventions.

The prototype was developed and runs on a Sun 3 and is directly transferable to a ruggedized Genisco computer.

CAPS structure. Figure 6 shows the three main components of CAPS: a user inter-

OVERVIEW OF PSDL

PSDL provides the designer with a uniform conceptual framework and a high-level system description. PSDL components are either operators or types, realized by decomposing PSDL or by retrieving or writing code in an underlying language.

As Figure A shows, PSDL decompositions are augmented computation graphs. The vertices (circles) are operators and the edges (lines) are data streams.

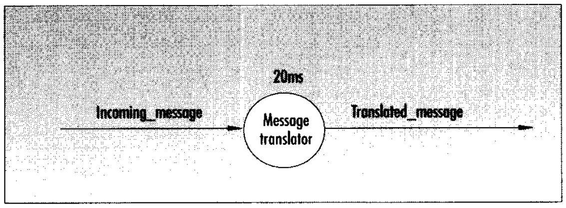


***Figure A.** Sample PSDL graph.*

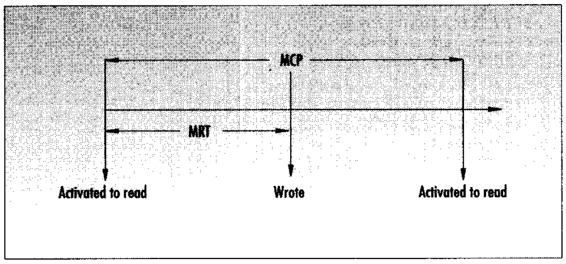


***Figure B.** Diagram of maximum response time (MRT) and minimum calling period (MCP) for a time-critical sporadic operator.*

Operators are state machines and their internal states are modeled by variable sets. Operators with an empty variable set behave like functions.

Data streams transmit data values from one operator to another. All the data values in a stream are instances of an abstract data type associated with the stream. Data types can be defined either in PSDL or the underlying language.

Data streams can be either dataflow streams or sampled streams. In PSDL, dataflow streams act as FIFO buffers of capacity one and synchronize data-driven computations. Dataflow streams guarantee that each data value written into the stream is read exactly once. Data values are removed from dataflow streams when they are read.

Sampled streams act as atomic memory cells and connect operators that fire at uncoordinated rates. Sampled streams model data sources for which only the most recent information is meaningful. Data values are removed from sampled streams when they are overwritten.

Constraints. Each vertex is augmented with a set of timing and control constraints.

Timing constraints. As Figure A shows, each vertex is labeled with a maximum execution time. The maximum execution time is the longest time between the instant an operator begins execution and the instant it completes execution. For example, in Figure A the message-translator operator takes as input a message and outputs a translated message in no more than 20 ms. Other timing and control constraints are expressed in text.

Operators can be triggered by data streams or periodic timing constraints. Operators triggered by data streams are called sporadic operators. In addition face, software database, and execution-support system.

User interface. The user interface, which supports concurrent tools, is implemented using InterViews,[8] which was developed at Stanford University and is based on X Windows (as is TAE Plus), so it is portable.

The user interface includes a graphics editor, a syntax-directed editor, and a tool interface. The graphics editor lets the designer edit a graphical representation of the prototype and automatically produces a PSDL representation that other CAPS tools can use.

The designer can specify parts of a prototype using graphical objects to represent PSDL computational structures like operators and data streams. The designer enters text annotations with the syntax-directed editor. The tool interface hides the details of the interfaces among CAPS tools from the designer.

Database. The software database, which includes a design and software database, holds reusable components and manages the configuration.

CAPS was not integrated with the software database when we conducted our experiment, so we used a simulated database of reusable Ada components to generate the C^3I prototype.

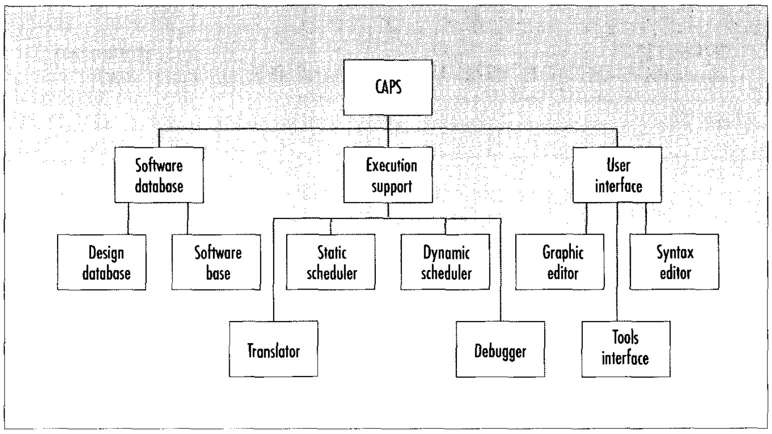


Figure 6. *Three main CAPS components and their associated tools.*

to a maximum execution time, each time-critical sporadic operator has a maximum response time and a minimum calling period, as Figure B shows.

The maximum response time is the longest time that may elapse between the instant an operator is activated to read its input streams and the instant it writes an event. The minimum calling period is the shortest time between two successive activations.

You can view the maximum response time as the operator's window of opportunity, the maximum execution time as the used portion of the window, and the minimum calling period as the maximum firing rate the system must support. The minimum calling period determines the amount of CPU time the system must allocate to the operator.

Operators triggered by periodic timing constraints are called periodic operators. Periodic operators are triggered by temporal events that must occur at regularly scheduled intervals. Figure C illustrates how the scheduling interval and deadlines are specified.

A periodic operator's execution must fit entirely within the scheduling interval, which is analogous to the maximum response time of a sporadic operator. You can view scheduling intervals as sliding windows whose position on the time axis relative to each other is fixed by a specified period and whose absolute position is fixed by the time the first read occurs, as Figure D illustrates. The first read must be scheduled less than one period after the system starts operation.

Control constraints. You use control constraints to adapt reusable code to particular designs. Control constraints can express conditional execution and output and control exceptions and timers. Triggering conditions and output guards are predicates.

If an operator is guarded by a triggering condition, the system discards input data that does not satisfy the condition without firing the operator. Output guards associated with an operator prevent computed output data from being written into the guarded streams if the condition is not satisfied.

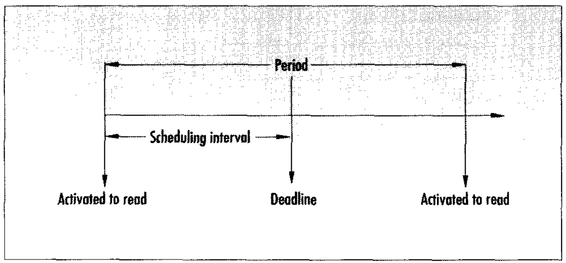


Figure C. *Diagram of how temporal events occur at regular scheduling intervals to trigger period operators.*

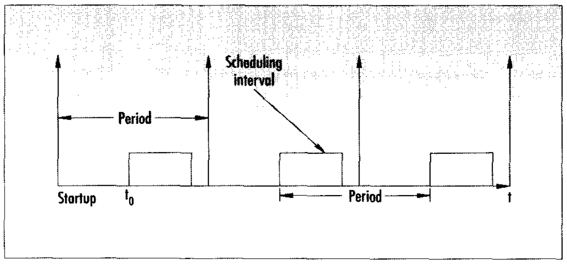


Figure D. *Scheduling interval.*

```
package TL is
  procedure MESSAGE_TYPE_DECISION_DRIVER;
  — Declarations of other driver procedures go here.
end TL;

  with SB; use SB;
  with PSDL_STREAMS; use PSDL_STREAMS;
  with DS_Debug_PKG; use DS_Debug_PKG;
  with PSDL_TIMER_PKG;
package body TL is
  type PSDL_EXCEPTION is
    (UNDECLARED_ADA_EXCEPTION);
  package C3I_SYSTEM_SPEC is
    package DS_COMMS_EMAIL is new
          FIFO_BUFFER(FILENAME);
    package DS_COMMS_TEXT_FILE is new
          FIFO_BUFFER(TEXT_RECORD);
    package DS_TDD_ARCHIVE_SETUP is new
          FIFO_BUFFER(ARCHIVE_SETUP);
    package DS_INPUT_TEXT_RECORD is new
          FIFO_BUFFER(TEXT_RECORD);
  — Other data stream declarations go here.
  end C3I_SYSTEM_SPEC;

  procedure MESSAGE_TYPE_DECISION_DRIVER is
    LV_INPUT_TEXT_RECORD: TEXT_RECORD;
    LV_TDD_ARCHIVE_SETUP: ARCHIVE_SETUP;
    LV_COMMS_TEXT_FILE: TEXT_RECORD;
    LV_COMMS_EMAIL: FILENAME;
    EXCEPTION_HAS_OCCURRED: boolean := false;
    EXCEPTION_ID: PSDL_EXCEPTION;
  begin
    if C3I_SYSTEM_SPEC.DS_INPUT_TEXT_
      RECORD.NEW_DATA then
      begin
        C3I_SYSTEM_SPEC.DS_INPUT_TEXT_
            RECORD.BUFFER.READ
                  (LV_INPUT_TEXT_RECORD);
      exception
        when BUFFER_UNDERFLOW =>
          DS_Debug.Buffer_Underflow
                  ("INPUT_TEXT_RECORD",
               "MESSAGE_TYPE_DECISION");
  end;
  begin
    C3I_SYSTEM_SPEC.DS_TDD_ARCHIVE_
      SETUP.BUFFER.READ
                  (LV_TDD_ARCHIVE_SETUP);
  exception
    when BUFFER_UNDERFLOW =>
      DS_Debug.Buffer_Underflow
                  ("TDD_ARCHIVE_SETUP",
               "MESSAGE_TYPE_DECISION");
  end;
  if true then
    begin
      MESSAGE_TYPE_DECISION
      (LV_INPUT_TEXT_RECORD,
        LV_TDD_ARCHIVE_SETUP,LV_COMMS_TEXT_
          FILE, LV_COMMS_EMAIL);
    exception
      when others =>
        DS_Debug.Undeclared_Exception
          ("MESSAGE_TYPE_DECISION");
        EXCEPTION_HAS_OCCURRED := true;
        EXCEPTION_ID := UNDECLARED_ADA_
          EXCEPTION;
    end;
    if not LV_COMMS_TEXT_FILE.IS_TRACK
    then
      begin
        C3I_SYSTEM_SPEC.DS_COMMS_EMAIL.
            BUFFER.WRITE
                  (LV_COMMS_EMAIL);
      exception
        when BUFFER_OVERFLOW =>
          DS_Debug.Buffer_Overflow
          ("COMMS_EMAIL",
          "MESSAGE_TYPE_DECISION");
      end;
    end if;
    if LV_COMMS_TEXT_FILE.IS_TRACK
    then
      begin
        C3I_SYSTEM_SPEC.DS_COMMS_TEXT_
          FILE.BUFFER.WRITE
              (LV_COMMS_TEXT_FILE);
      exception
        when BUFFER_OVERFLOW =>
          DS_Debug.Buffer_Overflow
          ("COMMS_TEXT_FILE",
              "MESSAGE_TYPE_DECISION");
      end;
    end if;
      if EXCEPTION_HAS_OCCURRED then
        DS_Debug.Unhandled_Exception
          ("MESSAGE_TYPE_DECISION",
            PSDL_EXCEPTION'
            image(EXCEPTION_ID));
      end if;
    end if;
  end if;
 end MESSAGE_TYPE_DECISION_DRIVER;
 — Other driver procedure declarations go here.
end TL;
```

***Figure** 7. Ada driver for the message_type_decision operator in Figure 5.*

We are now building the software database system, using existing object-oriented databases and formal models for prototyping design databases and software databases.[9]

Execution support. The execution-support system includes a translator, static scheduler, dynamic scheduler, and debugger.

♦ The translator generates code that binds the reusable components extracted from the software database. Its main functions are to implement data streams, control constraints, and timers.

♦ The static scheduler uses several algorithms to allocate time slots for operators with real-time constraints before execution begins.[10] If this allocation succeeds, all the operators are guaranteed to meet their deadlines even in the worst case. If the static scheduler can't find a valid

```
with GLOBAL_DECLARATIONS; use GLOBAL_DECLARATIONS;
with DS_DEBUG_PKG; use DS_DEBUG_PKG;
with TL; use TL;
with DS_PACKAGE; use DS_PACKAGE;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
with CALENDAR; use CALENDAR;
with TEXT_IO; use TEXT_IO;
procedure STATIC_SCHEDULE is
  MESSAGE_TYPE_DECISION_TIMING_ERROR : exception;
  — Other exception declarations go here.
  task type SCHEDULE_TYPE is
    pragma priority (STATIC_SCHEDULE_PRIORITY);
  end SCHEDULE_TYPE;
  for SCHEDULE_TYPE'STORAGE_SIZE use 200_000;
  SCHEDULE : SCHEDULE_TYPE;
  task body SCHEDULE_TYPE is
    PERIOD : duration := duration(5.00000000000000E+01);
    MESSAGE_TYPE_DECISION_STOP_TIME3 : duration :=
      duration(2.20000000000000E+00); — Deadline for message_type_decision.
    — Other declarations of scheduled stopping times go here.
    SLACK_TIME : duration;
    START_OF_PERIOD : time := clock;
    CURRENT_TIME : duration;

  begin
    loop
      begin
        — Calls on other driver procedures go here.
        MESSAGE_TYPE_DECISION_DRIVER;
        SLACK_TIME :=
            START_OF_PERIOD + MESSAGE_TYPE_DECISION_STOP_TIME3 -
              CLOCK;
        if SLACK_TIME >= 0.0 then
          delay (SLACK_TIME);
        else
          raise MESSAGE_TYPE_DECISION_TIMING_ERROR;
        end if;
        — Calls on other driver procedures go here.
        START_OF_PERIOD := START_OF_PERIOD + PERIOD;
        delay (START_OF_PERIOD - clock);
      exception
        when MESSAGE_TYPE_DECISION_TIMING_ERROR =>
          PUT_LINE("timing error from operator MESSAGE_TYPE_DECISION");
          START_OF_PERIOD := clock;
          — Other exception handlers go here.
      end;
    end loop;
  end SCHEDULE_TYPE;

begin
  null; — Initializations are not needed for this example.
end STATIC_SCHEDULE;
```

Figure 8. *Ada static schedule task generated for the message_type_decision operator in Figure 5.*

schedule, it provides diagnostic information about the cause of the problem and if it can be solved by adding more processors.

- The dynamic scheduler allocates time slots for operators that are not time critical.
- The debugger monitors timing constraints and various aspects of design integrity as the prototype runs, reports failures, and lets the designer adjust deadlines.

CAPS is being developed as an ongoing research effort, and some of the functions just listed were not ready when we started our experiment.

When we started decomposing the modules for the C^{3}I station, the graphics and syntax-directed editors were not ready to use for a multilevel PSDL example, so we used Frame Technology Corp.'s Framemaker to draw the graphs and write the PSDL code.

After completing the multilevel decomposition, we prepared a PSDL file that included only the atomic operators in the bottom level of the decomposition. We did the constraint propagation and consistency checking among levels and modules manually.

Prototyping steps. Generating a prototype in CAPS has 11 general steps:

1. The designer draws the computation graphs with the graphics editor.
2. The graphics editor provides the skeleton PSDL code and propagates inherited constraints.
3. The designer uses the syntax-directed editor to modify the skeleton code, and the system produces a file with the prototype's PSDL description.
4. The translator produces an Ada package that instantiates the data streams, reads data from and writes data to the data streams, and executes atomic operators. The translator uses PSDL descriptions to generate driver procedures for atomic operators. For example, Figure 7 shows the Ada driver procedure for the PSDL message_type_decision operator in Figure 5.

The driver procedures provide a standard interface between the Ada components and the generated scheduling software. They include exception handlers for stream overflow and underflow conditions and for undeclared exceptions that might be raised by faulty implementations of atomic Ada components. The exception handlers interface to the PSDL debugger to produce diagnostic messages.

5. The static scheduler tries to find a schedule for the time-critical operators and — if it finds a feasible schedule — produces an Ada package that contains the schedule, represented as an Ada task that calls the driver procedures. Figure 8 shows the part of the static schedule task gener-

```
with TL; use TL;
with PRIORITY_DEFINITIONS; use PRIORITY_DEFINITIONS;
package DS_PACKAGE is
  task type DYNAMIC_SCHEDULE_TYPE is
    pragma priority (DYNAMIC_SCHEDULE_PRIORITY);
  end DYNAMIC_SCHEDULE_TYPE;
  for DYNAMIC_SCHEDULE_TYPE'STORAGE_SIZE use 100_000;
  DYNAMIC_SCHEDULE : DYNAMIC_SCHEDULE_TYPE;
end DS_PACKAGE;

package body DS_PACKAGE is
  task body DYNAMIC_SCHEDULE_TYPE is
  begin
    delay (1.0);
    loop
      STATUS_SCREEN_DRIVER;
      MESSAGE_EDITOR_DRIVER;
      — Invocations of other non-critical operators go here.
    end loop;
  end DYNAMIC_SCHEDULE_TYPE;
end DS_PACKAGE;
```

Figure 9. *Ada task to invoke noncritical operators.*

ated for the PSDL description in Figure 5.

The static schedule contains time allocations for the time-critical operators in a fixed pattern that can be repeated indefinitely. The static scheduler determines the length of this pattern, which is represented by the Ada constant Period. The schedule also includes a control structure that monitors time-critical components and reports missed deadlines, which are determined by the static scheduler and are represented by Ada constants like message_type_decision_stop_time3. The static scheduler recovers from missed deadlines by resetting its time reference and skipping to the next iteration of the static schedule.

6. Once the static schedule is found, the dynamic scheduler produces an Ada package that contains a dynamic schedule for noncritical operators. This task, shown in Figure 9, invokes noncritical operators during time slots not being used by the static-schedule task. Unused time slots can arise because of either scheduled waiting periods or an operator's early completion. Relatively large vacant slots can be created when PSDL control constraints suppress the execution of a time-critical operator for a subset of all the potential activations.

The dynamic schedule is represented as an Ada task with a priority less than that of the static schedule task, so it can be executed whenever there is nothing more important to do. This decouples the analysis of the time-critical operator's resource requirements from the design and implementation of the prototype's noncritical parts, thus simplifying the analysis and speeding prototyping.

Context switching is handled by the scheduling mechanism provided by the Ada runtime system and does not require any special code to be generated, other than the pragmas that declare the priorities of the schedule tasks.

7. CAPS provides the designer with matching reusable Ada components for the atomic operators. If a reusable component cannot be found, the designer either writes the code for that operator or decomposes it in an effort to find reusable components. (We are now designing a tool that can generate Ada code from equations describing the desired behavior.)

8. CAPS compiles and loads the code and begins executing the prototype.

9. Potential users observe the prototype's behavior, paying particular attention to the consequences of arbitrary assumptions.

10. The designer modifies the prototype in response to user feedback.

11. When users accept the prototype's demonstrated behavior, the designer adds any required noncritical functions, optimizes the prototype, and ports it to the target hardware and operating system.

LESSONS LEARNED

We used CAPS to successfully generate an Ada C^3I prototype quickly and at low cost. The prototype was constructed with about one man-month of effort, not counting time spent in formulating the requirements and fixing problems with the tools.

The resulting Ada prototype executes in a color, graphical, multiwindow user interface; provides all essential functions defined in the prototype specification; and proves that all the hard real-time constraints placed on the station's components are met completely.

Bus errors. During prototype execution, the system continuously gave bus errors at a certain point. After a long debugging effort, we noticed that the error occurred only for the data stream defined by the last stream declaration in the Ada package in Figure 8. We solved this error by adding an extra stream that the program did not use. Although we could not find any reason for it, we suspect the problem was caused by a compiler fault.

Another problem during execution involved the schedulers. Because the prototype uses so many variables, the default storage for the static and dynamic schedule tasks was not large enough. So we modified the static and dynamic schedulers to generate Ada code that explicitly allocates more storage via representation clauses. During the experiment, we used a constant for the storage size. To reduce portability problems, we are investigating the design of an enhancement that will calculate the required storage based on actual variable use and the size attribute provided by Ada.

Relative speeds. While the timing constraints are feasible for a stand-alone Sun workstation of the type proposed for the final system (a Sun SparcStation), this hardware was not available to us. Our prototype was designed on an older Sun system, which is much slower than the proposed hardware.

This forced us to use longer maximum execution times and periods to make the prototype run. We learned that the prototype need not execute as fast as the requirements specify, but rather must meet the requirements relative to the speed of the proposed target hardware.

This realization focused our research on better methods for evaluating the feasibility of real-time constraints when the target hardware for the proposed system differs from the prototype's hardware[4] and it has resulted in changes to the design of the CAPS system to support explicit resource models for the target hardware.

Global constraints. The prototype does not address global timing constraints because the version of CAPS we used did not support a multiprocessor model. We are working on ways to realize global timing constraints in distributed multiprocessor systems with bounded communication delays in point-to-point data transmissions.

Any design that guarantees global message delivery within hard real-time constraints depends on bounded delivery times for the long-haul network, at least for transmissions between nodes that are directly connected. However, such networks are impossible to realize because in practice you must also guarantee accurate message delivery. If the underlying medium is noisy — which is likely in C^3I applications because of jamming — designs that guarantee bounded message delays must tolerate some message loss. That's because error-correcting protocols can retransmit only a bounded number of times if the transmission delay is limited by hard real-time constraints.

Retransmission can reduce the message-loss rate, but if a message can get lost in a single transmission, it can also get lost in n consecutive transmissions. We should therefore bound message-delivery time by a constant times the required retransmissions and limit the retransmissions that can be attempted before a time-out error must be reported.

The capabilities CAPS provides are essential to rapid prototyping. In particular, automatic code generation and instrumentation let us try design variations quickly without cutting corners because the diagnostics helped us localize and fix bugs. Automated schedule construction and diagnostic information about timing constraints helped us navigate the maze of interacting resource constraints and evaluate the feasibility of the requirements.

The experience we gained also suggests many improvements to CAPS. For example, before our experiment the CAPS static scheduler required the designer to specify a maximum response time and a minimum calling period for each time-critical sporadic operator. We found that the designer often did not know these two attributes, so we modified the static scheduler to calculate default values based on heuristics that seek the fastest feasible responses yet maintain a balanced use of resources.

The C^3I prototype is also serving as a testbed for ongoing research in computer-aided software design. A hypothetical network of generic C^3I stations is serving as a test case to investigate deadlock detection and prevention at the design level. The goal of this research is to develop a tool that takes as input a formal specification of a distributed system, determines if the design makes deadlock possible and if so, guides the designer in removing that possibility. ◆

Luqi is an associate professor at the Naval Postgraduate School. Her research interests include rapid prototyping, real-time systems, and software-development tools.

She received a BS from Jilin University, China, and an MS and a PhD in computer science from the University of Minnesota.

Address questions about this article to Luqi at Naval Postgraduate School, NPS 052, Monterey, CA 93943; Internet luqi@cs.nps.navy.mil.

ACKNOWLEDGMENTS

I thank Jeff Schweiger, Gary Hughes, Valdis Berzins, Steve Anderson, Cengiz Kesoglu, and Vedat Coskun for their contribution to this research and the anonymous referees who helped me improve this article.

REFERENCES

1. W.R. Beam, *Command, Control, and Communications Engineering*, McGraw-Hill, New York, 1989.
2. Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software," *IEEE Trans. Software Eng.*, Oct. 1988, pp. 1409-1423.
3. Luqi and M. Ketabchi, "A Computer-Aided Prototyping System," *IEEE Software*, March 1988, pp. 66-72.
4. Luqi, "Software Evolution Through Rapid Prototyping," *Computer*, May 1989, pp. 13-25.
5. F.S. Anderson, *Functional Specification For a Generic C^3I Station*, master's thesis, Computer Science Dept., Naval Postgraduate School, Monterey, Calif., 1990.
6. V. Coskun and C. Kesoglu, *A Software Prototype for a Command, Control, Communications, and Intelligence (C^3I) Workstation*, master's thesis, Computer Science Dept., Naval Postgraduate School, Monterey, Calif., 1990.
7. *Transportable Applications Environment Plus*, Nat'l Aeronautics and Space Admin., Goddard Space Flight Center, Greenbelt, Md., 1990.
8. M.A. Linton, J.M. Vlissides, and P.R. Calder, "Composing User Interfaces with InterViews," *Computer*, Feb. 1989, pp. 8-22.
9. E. Borison, "Program Changes and Cost of Selective Recompilation," Tech. Report CMU-CS-89-205, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, 1989.
10. J. Stankovic and K. Ramamritham, *Hard Real-Time Systems Tutorial*, IEEE CS Press, Los Alamitos, Calif., 1988.