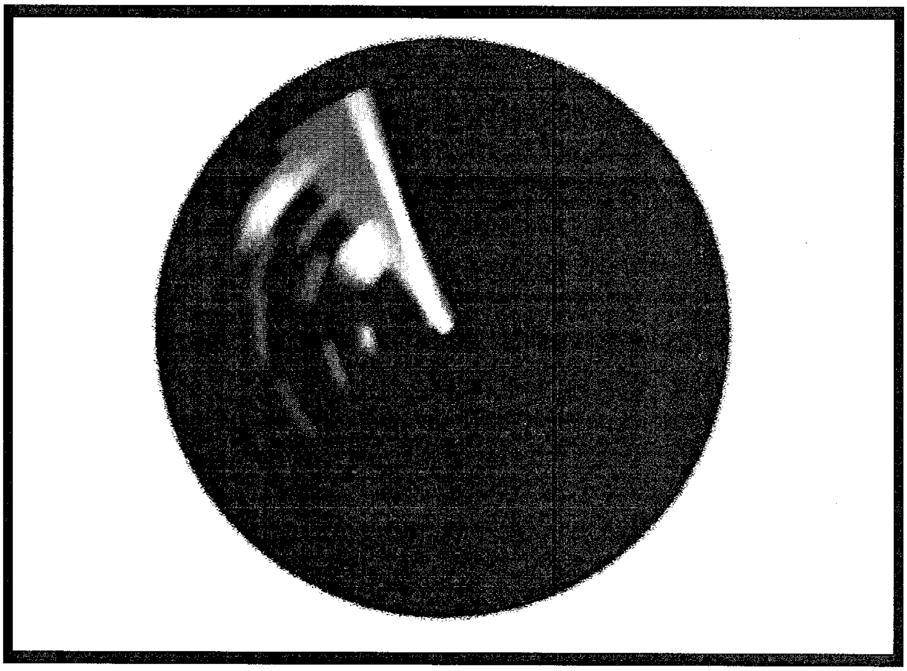
Using Formal Methods to Develop an ATC Information System



Can formal methods be a part of large-system development? The project teams at Praxis used a combination of formal methods to help specify, design, and verify CDIS, a large information-display system within an ATC-support system. Their project suggests that it can be practicable *and* beneficial.

ANTHONY HALL, Praxis

To handle increasing traffic, the UK is in the process of upgrading its air-traffic-management system. The upgrade includes developing the Central Control Function, a new way to handle terminal traffic. The CCF provides automated support for controllers in the London Area and Terminal Control Centre, including *approach sequencing*, a function for generating and manipulating the inbound-flight sequence to major airport complexes such as Heathrow or Gatwick. The automated support is handled by several systems, including an upgraded National Airspace System, a new radar system, an Airport Data Information System, a new digital closed-circuit television, and a new information system: the CCF Display Information System, which we developed at Praxis and delivered to the Civil Aviation Authority in 1992.

CDIS is responsible for displaying information to controllers about arriving and departing flights, weather conditions and equipment status at airports, and other support information provided by CDIS data-entry staff. It also maintains real-time displays of its own status to allow the engineers to control the system.

We used formal methods in the specification, design, and verification of CDIS, making it one of the largest applications of formal methods attempted thus far. Because of the system's size and complexity, we used several formal methods to develop its sequential and concurrent aspects, in some cases combining formal with more conventional methods. We also used different notations at different project stages, both for technical rea-

0740-7459/96/$05.00 © 1996 IEEE

sons, such as when one notation gave us better modularity, and nontechnical reasons, such as to suit a particular team's expertise.

CCF AND CDIS

In the terminal control room at LATCC, air traffic controllers use about 30 controller workstations to display information and manipulate the approach sequence. Other operational staff receive display information from 20 simplified controller workstations. The CCF supervisor, engineers, and data-entry staff operate six administrative workstations. Each controller workstation uses a pair of computers; if one fails, processing switches to the standby machine.

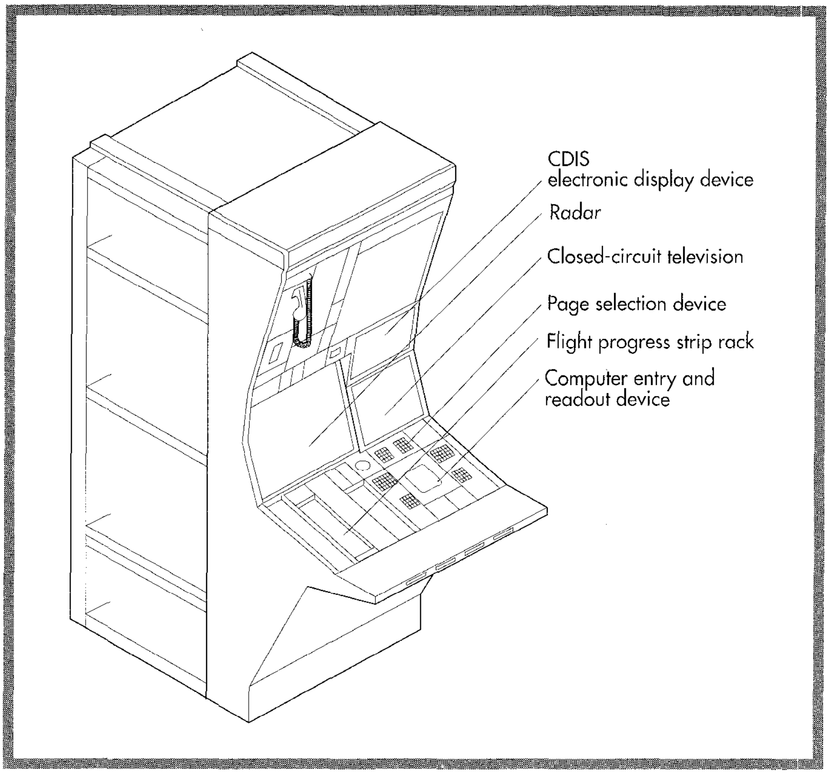

CDIS
electronic display device
Radar
Closed-circuit television
Page selection device
Flight progress strip rack
Computer entry and readout device

Figure 1. The CCF controller's workstation.

Figure 1 shows a CCF controller's workstation. The main CDIS display is a 19-inch color-graphics screen that displays pages of information. Pages are selected using the page-selection device, a custom-built keypad that allows one-keystroke selection of frequently used pages and provides extra keys for special functions such as acknowledging changed data and reading broadcast messages. The controller who managers the inbound-approach sequence — the approach-sequence allocator — uses the computer entry and readout device, a touch sensitive plasma display, to directly manipulate the landing order of flights.

CDIS receives information in real time through an X.25 network connection to the National Airspace System and the Airport Data Information Systems at the major airports. It also has its own store of information, produced and edited internally using administrative workstations. CDIS sends information to the closed-circuit television system over an X.25 link.

Requirements. Because CDIS is a real-time, operational system, it has stringent performance and availability requirements: Information must be displayed within one or two seconds of receipt, the system must be available at least 99.97 percent of the time, and there can be no single point of failure.

These requirements, shown in Table 1, were the basis of our development methods and our design of CDIS. Figure 2 shows the CDIS hardware architecture. The CDIS Central Processing System is a fault-tolerant computer that acts as the point of communication with all external systems, provides the central repository of CDIS data, and controls the system — including failure management and workstation recovery. Communication between workstations and the CCPS is through a LAN consisting of two token rings that act in main/standby mode.

CDIS development. We developed CDIS in two stages. In 1989, we defined the requirements; we then began the implementation project, which ran through 1992. Upon delivery, the Civil Aviation Authority integrated CDIS with other CCF systems. The system went into operational use in the autumn of 1993.

The software-implementation project had five major phases: system specification, software design, coding and unit test, system test, and acceptance test. Coding and testing were done in parallel by two separate teams. One team coded and unit tested the software; the other integrated the software and did black-box system testing and acceptance testing. We developed the software in six incremental builds, each one a formal handover from the implementation team to the test team.

We carried out acceptance testing by running a subset of the system tests — agreed on with the client — that covered all aspects including each functional area, all performance requirements, and full resilience testing.

Fault management is central to the development and maintenance of CDIS. Fault management began when the implementation team delivered the software to the test team, and continued through operational system integration and into maintenance. We analyzed each fault to determine its point of introduction. If, for example, the specification was wrong, then the spec-

TABLE 1
CDIS REQUIREMENTS

Requirement	Hardware	Software	Development Method
Functionality			Formal specification, prototyping, formal design
Performance	Distributed processing, multi-processor hardware	Data located at point of use, optimized process organization	Performance calculation
Availability	Dualed hardware, hardware health monitoring	Fail-stop processes, defensive programming	Formal design, precondition checking, proving critical properties

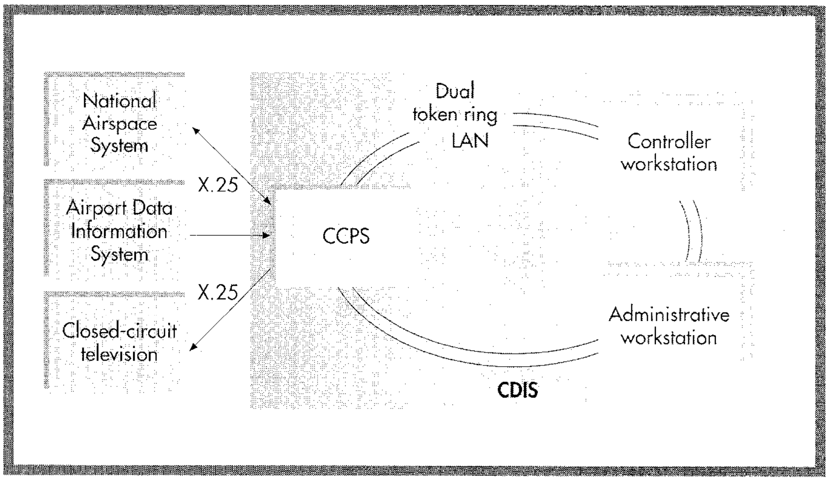

National Airspace System
Airport Data Information System
Closed-circuit television
X.25
X.25
Dual token ring LAN
CCPS
Controller workstation
Administrative workstation
CDIS

Figure 2. CDIS architecture.

ification, design, and code were all fixed. This meant that the CDIS specification, design, code, and user documents were always in step — and we thus avoided the common situation in which the code is the only real authority on what the system does.

FUNCTIONAL REQUIREMENTS

We used three techniques to develop CDIS functional requirements. First, we built a world model using entity-relationship analysis to describe the real-world objects that CDIS had to deal with, the properties of these objects, and the relationships between them. Second, we defined the processing requirements using dataflow diagrams following a real-time structured-analysis method.[1]

Finally, we used VDM,[2] a formal-specification language, to give a precise definition of the data maintained by CDIS and the operations on the data. In addition to being technically suitable, VDM was familiar to the requirements team and had been used on other projects for the Civil Aviation Authority. At this stage, the VDM model was only partial. It included the major data structures and some of the most critical operations.

Formal and semiformal notations. During the study, our ideas of how to use formal specification changed. Initially, we expected to define system operations using the dataflow diagrams to represent the processing for each operation, and then to use formal specifications to define the lowest level processes on these diagrams. This is similar to the method suggested by Nico Plat and his colleagues.[3] However, we found that this approach did not give useful results for two reasons. First, decomposing an operation into lower level processes does not really specify the operation at all — rather, it sketches its design. Second, specifying the processes on a dataflow diagram does not clearly specify the diagram as a whole, because the meaning of the dataflows is left undefined. Because of this, we decided to carry out the formal specification at the top level.

We defined a VDM state that represented the whole state of the CDIS system, and individual VDM operations that corresponded to individual user-level operations — *events* in the structured model. For example, receipt of a message from an external system is an event with a corresponding VDM operation. One part of the structured dataflow model is still primary: the *context diagram*. This shows CDIS as a single process and identifies the dataflows with all the external systems and users of CDIS. It thus defines *what* operations are needed, along with their inputs and outputs. However, we did not use dataflow diagrams at all to define the *effects* of the operations.

The state in the VDM specification is closely related to the entity-relationship model. Although it is possible, in principle, to derive VDM state directly from an entity-relationship model, VDM is richer and thus it is better to *replace* some entity-relationship constructs by simpler and more direct formal representations. We also made the model more precise by adding more detailed constraints than the simple cardinalities

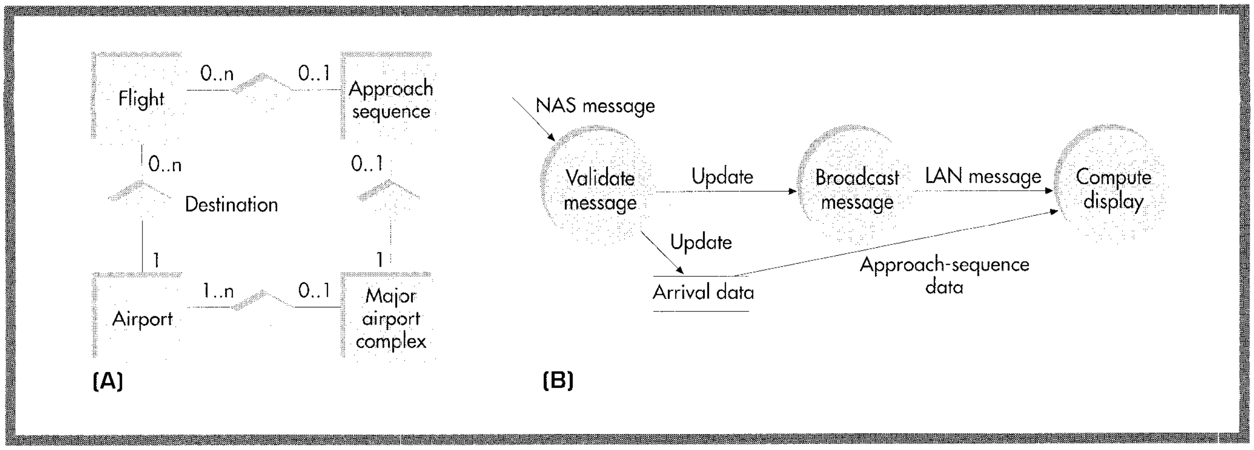


Figure 3. *(A) ER fragment showing approach sequence. (B) Dataflow diagram for approach-sequence update.*

of the entity-relationship model.

Specification types. The following example illustrates the different kinds of specification. One of the CDIS functions is to receive messages from the National Airspace System about the approach sequence and to update the displays with the new information. The corresponding piece of state is shown as an entity-relationship diagram in Figure 3a, and Figure 3b shows a typical dataflow diagram for the operation. Figure 3b shows that a NAS message is first validated, and then the data from the message is stored in the *Arrival data* store. An update is then sent to a process that broadcasts a message over the LAN, and this message, along with the stored arrival data, is used to compute the display's new value.

These two diagrams are typical of structured requirements analysis. However, the dataflow diagram in particular is unsatisfactory as a requirement statement. First, it is unclear, for example, what the validation process does or what happens if it fails. Second, the fact that a message is broadcast, and that display computation uses stored data rather than the LAN message contents, are design decisions. They are of no interest to the user and, in any case, it is premature to assume an implementation strategy at the requirements stage. Finally, if we specify the processes on the diagram, we find that the resulting low-level specifications do not help the user understand the overall effect of processing the message.

VDM specification. To write a VDM specification of the operation, we first modeled the state from the entity-relationship diagram in VDM. The main notations used in the VDM specification (apart from the usual logical and set-theoretic symbols) are:

- **state**: declaration of state components,
- **inv**: constraints on the state,
- **operations**: operations on the state,
- **ext**: external state affected by an operation,
- **post**: postcondition defining effect of operation, and
- $\overleftharpoon{data}$: value of *data* before operation,

Figure 4a is a simplified version of the VDM state.

The mathematics in Figure 4a correspond closely to the entity-relationship diagram in Figure 3a. The use of functions and the constraints on their domains and ranges represent the cardinality constraints expressed in the entity-relationship diagram. However, the VDM is an improvement because:

- it represents the approach sequence directly by a *sequence* of flights associated with the major airport complex (the ordering of flights in the sequence is not expressed in the entity-relationship diagram at all), and
- it expresses that, for a flight to be in the approach sequence for a major airport complex, it must be headed for an airport associated with that complex (although not all such flights need be in the sequence). This fact is not expressible in entity-relationship notation.

To define the effect of an update message we need more state, because we must represent the information about each flight and the displays. Figure 4b shows that the definition of this operation can be understood in small pieces, each of which is understandable in terms of user concepts. Validation is encapsulated in the definition of *can_update_arrival_data*, the way the data are updated is encapsulated in *arrival_data_updated*, and the effect on the displays is encapsulated in the function *arrival_data_displayed*. Each function is defined elsewhere in the specification. For example, *arrival_data_displayed* defines how a given collection of arrival data appears on the screens. It is defined in terms of a lower level function that describes how arrival data appear on a particular page; that, in turn, is based on the internal definition of how a page is set up, and encapsulates the rules for how arrivals are selected and ordered for display.

Findings. We used VDM as part of requirements analysis because we believed that its precision would both help clarify our understanding of the requirements and make them complete and unambiguous. This belief was borne out in practice. During the study we asked a lot of questions — many of them a result of trying to formalize certain requirements — and thus gained a good understanding of what the system was intended to do. However, we also encountered several problems.

Whether formal or informal, this kind of functional specification cannot distinguish essential requirements from those that are merely desirable. In addition, the first-order logic lets us write

```
state
airports:set of Airport
flights:set of Flight
MACs:set of MAC
airport_MACs: map Airport to MAC
approach_seqs:map MAC to seq Flight
destination:map Flight to Airport

inv
dom destination = flights ∧
ran destination ⊆ airports ∧
dom airport_MACs ⊆ airports ∧
ran airport_MACs = MACs ∧
dom approach_seqs ⊆ MACs ∧
∀ m ∈ dom approach_seqs •
   elems approach_seqs(m) ⊆
       {f ∈ flights | airport_MACs(destination(f))= m}
(A)

state
arrival_data:map Flight to Data
edd_displays:map EDD to EDD_display
edd_pages:map EDD to Page

operations
AS_MSG(msg:Approach_Sequence_Message)
    ext
        wr arrival_data
        wr approach_seqs
        wr edd_displays
        rd edd_pages
    post

    if can_update_arrival_data(msg, arrival_data)
    then arrival_data_updated(msg, arrival_data, arrival_data,
                approach_seqs, approach_seqs) ∧
        arrival_data_displayed(edd_pages, arrival_data,
                approach_seqs, edd_displays)
    else arrival_data = arrival_data ∧
        approach_seqs = approach_seqs ∧
        edd_displays = edd_displays
    fi

(B)
```

Figure 4. (A) A simplified version of the VDM state. (B) Operational specification in VDM.

down properties of each individual operation, but requirements are often expressed in more global terms as properties of *all* operations — for example, that all operations must be reversible. Such higher order properties cannot be directly expressed using VDM.

Similarly, any specification based on a system model and a specific set of operations has already committed to a considerable level of detail. The user might like to specify, for example, that there should be a complete set of query operations without having to specify what form these operations might take. Finally, only functional requirements can be captured using this kind of specification; usability, performance, reliability, and aspects of safety are all outside its scope.

SYSTEM SPECIFICATION

At the beginning of implementation, we decided to produce a complete specification of CDIS to serve as a basis for design. Because of our experience in the requirements study, we abandoned the use of dataflow diagrams and based the specification primarily on the formal notation. However, a specification entirely in VDM would not have been adequate for two main reasons:

♦ VDM provides no real help in specifying the user-interface details — one of the most important aspects of CDIS.

♦ The VDM specification describes only sequential aspects of behavior. CDIS processes many inputs concurrently, and we had to know which operations could occur concurrently and how they might interfere with each other.

We therefore produced the system specification in three parts: a formal *core specification*, a set of *user-interface definitions*, and a *concurrency specification*. The specifications constituted three different views of the system, rather than specifications of three different system parts. Figure 5 shows the relations between the views. The core specification described the data managed by CDIS and every operation that CDIS could perform. Each operation was specified at a semantic level by defining its inputs, outputs, and effect on the state. For each operation in the core specification, there was a corresponding user-interface definition that described the dialogue needed between user and machine to effect the operation, the required keystrokes or mouse actions, and the screen's appearance during the dialogue. The concurrency specification showed what real-world processes could carry out the operations. It also identified the data accesses and possible operation sequence of each process.

Core specification. Because CDIS has about 150 specification-level operations, we faced the problem of how to structure the specification into understandable modules. The top-level modules are related to major areas of functionality: arrivals, departures, airports, displays, and page, communications, and engineering management. Unfortunately, these modules are not independent and typical operations affect the state of several modules. For example, certain errors on the external links cause communications with the

National Airspace System to be lost; portions of the arrivals database thus become invalid and result in changes on controller displays. To express this type of behavior clearly, we needed a modularization mechanism that let us write operations affecting the state of several modules in a natural way.

We considered three alternatives: VDM, which did not have a useful modularization mechanism; Z, which seemed to offer the kind of structuring we needed;[4] and VVSL, a language with a VDM-like syntax and a well-developed module structure that provided most of the facilities we needed.[5] Because we had used VDM during the requirements analysis, we were reluctant to switch to Z — even though we considered its schema calculus ideal for the kind of modularization we wanted. Furthermore, Z's error-handling conventions are clumsy compared with those of VVSL. We therefore decided to use VVSL.

When one VVSL module imports another, it imports the constructs that are exported from the imported module. Unfortunately, there is no way of importing operations in this way because, unlike Z, VVSL has no calculus for combining operations. This caused us problems in building up the specification. For example, receipt of an arrival-update message affects the state in the arrivals and displays modules. We wanted to write the update as the conjunction of an operation `update_arrival_data` in the `arrival_data` module and an operation `update_arrival_displays` in the various `display` modules. However, in VVSL this was impossible, so we had to write two functions: *can_update_arrival_data* and *arrival_data_updated.* The first corresponds to the error checking of `update_arrival_data`, the second to the actual updates that would be carried out. The operation in the top-level module then uses these two functions to describe the effect in the arrivals module. This convention is the one that was used in Figure 4b. It unfortunately led to clutter in the specification, and operations were often not defined where we expected to find them. This made the specification less readable than we would have liked.

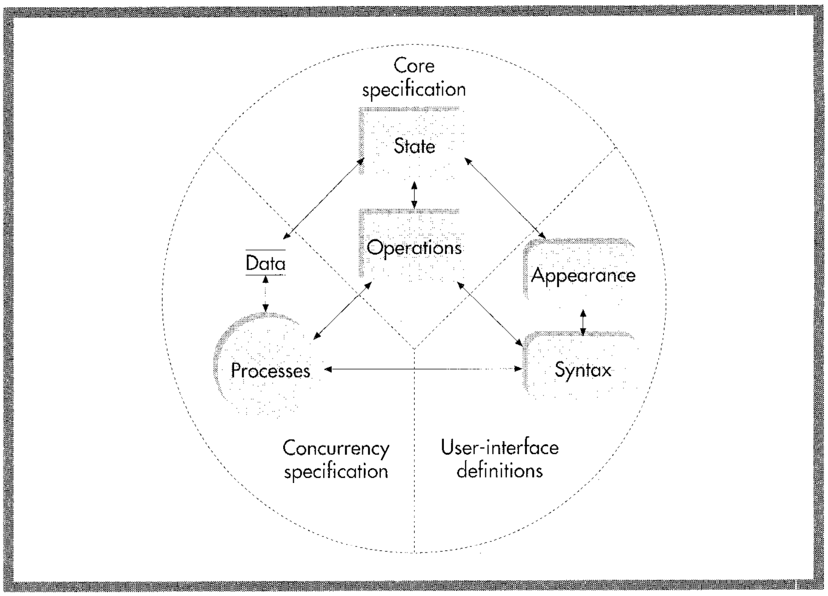


Figure 5. *The three CDIS specifications.*

The only tools we used in preparing the core specification were some document-preparation macros for LaTex. At the time, the only type-checking tools available would not let us generate easily readable output, nor would they support VVSL and its module constructs. If a suitable type-checker had been available, we could have eliminated many syntactic errors in the specification. On the other hand, we made several extensions to the notation that let us write the specification in a more compact and understandable way, and tools might have prevented us from doing that.

User-interface definitions. The user-interface definitions contained two kinds of information: the interface's physical appearance using pictures and text and the syntax of the user dialogues using state-transition diagrams. The definitions were developed using throw-away prototypes.

Each operation described at an abstract level in the core specification had a concrete specification in a user-interface definition. However, the level of abstraction was not uniform; it depended on the operation's complexity and importance. At one extreme, there is a single operation in the core specification, `Edit_Page`, that corresponds to a complete document, the Editor's User-Interface Definition. At the other extreme, the operations in the core specification describing how the approach-sequence allocator manipulates flights are at the level of individual keystrokes, because the behavior of this interface is critical and subtly related to the information being received from the National Airspace System while the operations are in progress.

Concurrency specification. The concurrency specification describes the *inherent* concurrency of the CDIS environment — the processes that could run concurrently in the real world. There was one such process for each CDIS user-input device, each connection to an external system, and each CDIS hardware device that was monitored for failure and recovery.

We used two notations for the concurrency specification. Each process was defined in the language of CSP.[6] The alphabet of each process was the set of VVSL operations available to it. We thus checked that the concurrency specification and core specification were consistent.

We also drew dataflow diagrams

showing all the processes and all the state variables they read and wrote. These diagrams, unlike those used in structured analysis, had a precise semantics: a circle represented a concurrent process, a data store represented a state variable, and dataflows represented read and write access to state variables by processes. This identified the shared data that needed to be accessed by several processes simultaneously.

Findings. Overall, I believe the specification phase was successful. It defined the functionality of CDIS precisely and provided a firm foundation for the project. Although the techniques we were using had not been used before in this combination or on this scale, we understood their strengths and limitations and had no major unexpected difficulties.

However, the specification is far from perfect; there are problems remaining and lessons to be learned. First, it is hard to get a system overview from the specification. The formal specification is not top-down — indeed, because we followed a declaration-before-use style, it is almost completely bottom-up. We also failed to write enough English commentary in the core specification, which makes it even more difficult to read.

We also faced the inevitable problems at the boundaries between the three kinds of specification. The user-interface definitions were not as complete as the core specification because the concrete representations in the user-interface definitions were (necessarily) *examples* of the appearance, not exhaustive definitions. For example, we failed to specify the colors of all aspects of the engineer's display under all circumstances, leading to uncertainties as to what the behavior was meant to be. We also did not separate concerns as much as we would have liked. During the design phase, for example, our idea of the user-interface changed from an "inquire and get a snapshot" style to a continuous-update style. This had a major effect on the core specification that we would have preferred to avoid.

We also had difficulty choosing the right level of detail. For example, we did not specify the validation of messages well and some of the internal-consistency checking — which should have been part of the specification — was left to the design phase.

The specification is far from perfect; there are problems remaining and lessons to be learned.

Finally, the specification is only an approximation of the real CDIS behavior. At the project's outset we made the simplifying assumption that operations are atomic. In practice, however, an operation can take several seconds to execute, giving ample time for other operations to interfere. As a result, there are observable states of CDIS — such as when only some screens have updated in response to a message — which are not allowed by the specification. Intuitively, it is obvious that these deviations from the specification are "harmless," but it is not clear how to decide what is and what is not allowable behavior. Further work on refining shared systems may illuminate this question.[7]

Writing the specification was a useful exercise in its own right. The process of defining the functionality and reviewing it with the Civil Aviation Authority helped us clarify requirements precisely. The formality of the notation certainly helped us focus this process.

Once complete, the specification was used for change control. Like any large project, CDIS suffered many requirements changes during its implementation. The existence of a clear and complete definition was invaluable in helping us decide what was and was not a real change, and in evaluating the impact of proposed changes.

The specification was the basis for design, implementation, and testing. For testing, the system specification superseded the original functional requirements, because it included everything they contained and more. The system tests were derived by the test team from the system specification and the nonfunctional requirements. In particular, the formal specification was used to derive black-box tests by equivalence partitioning and boundary value analysis in a systematic way. Every test was traced back to the part of the specification that was being tested, giving complete visibility of test coverage.

Finally, the specification served as a basis for the user documentation, which was, inevitably, prepared before the system was complete. Again, the specification's precision and completeness minimized the extent to which the user documentation had to be revised once the real system was available.

DESIGN

As Figure 6 shows, the CDIS design consisted of an overview and four kinds of design components. There is a difference between partitioning the design and the specification. Although the three kinds of specification were three different *views* of the same thing, the four different designs — functional, process, user-interface, and infrastructure design — were designs for different *parts* of the software. Thus, the relationship between the design components reflected both a division of the software into subsystems and a division of effort between teams. Also, because the different design components had different issues and required different programming styles, we used different notation in each of the four kinds of design. We used formal methods in designing the application modules, the processes, and the LAN software.

Functional design. To design the application modules, we devised a module structure for the application code and then specified the operations and internal data for each module. The module structure was derived from the core-specification structure, with two modifications. First, each module was usually divided into two layers — an operation layer, used by the process or user-interface software, and a services layer, used by other application modules. Each module was then divided into three parts for CCPS-specific processing, workstation-specific processing, and common code.

Our basic idea was to specify the application modules in VVSL as refinements of the corresponding specification modules. That is, the specification data types would be transformed into more concrete, computer-oriented types, and the operations redefined to use these types.

At first, we planned to use VVSL only for the more critical modules and to develop those quite formally, writing down the refinement relations and the corresponding proof obligations. For less critical modules, we planned to write more code-oriented specifications. Neither of these intentions was carried out in practice. First, it proved awkward to use different notations, so we decided to specify almost everything in VVSL (for simple cases we did not give the full specification of each operation, only its signature). Conversely, it turned out that the refinement relations were extremely large and cumbersome to write down, and we abandoned the attempt to establish a fully formal connection between the specification and the design. One problem was the large state-space of CDIS, in which each operation affected a large part of the state.

There was also a more basic reason we were unable to write down the refinement relation. Although the set of data in a design module was a concrete form of the data in the corresponding specification modules, the design module as a whole was not a refinement of the specification module, because it usually had a completely different set of operations. The specification of a user operation was not refined by a single operation on an application module; rather, it was implemented by a *transaction* that involved process code and user-interface code as well as operations in several application modules.

For example, Figure 7 shows the transaction involved in a single user operation to set an airport data value. No single operation in the airport design modules corresponds to the specification operation — instead, the module offers services that allow the user interface to check the validity of the operation, to send a request to the CCPS to make the change and broadcast the new data, an operation to accept the new data at the workstation, and many query operations to allow display of the new data. It is an open question how, formally, these operations are related to the formal specification.

Process design. Process design included the design of processes, tasks, interprocess communication, and data sharing. We derived the design primarily from the concurrency specification, along with the nonfunctional requirements — especially performance — and the characteristics of the machines and operating systems we were using. We began by identifying the processes and

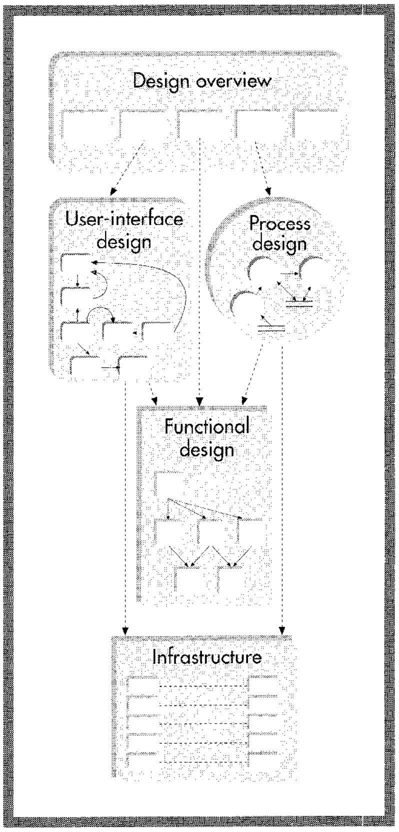


Figure 6. CDIS design components.

their communications mechanisms and shared data, and documenting these using dataflow diagrams. We then defined the response of each process to each possible event.

We used finite-state machines to diagram the process design. The only novel thing we did was to use VVSL predicates as a way of characterizing complex states. Also, the actions in the finite-state machines were application-module operations, specified in VVSL. In the more complex machines, we had difficulty relating the machines back to the specification they were implementing. In retrospect, encoding the finite state machines in VVSL might have been a better approach, giving us a better integration with the module specifications.

User-interface design. We derived this design from the user-interface definitions and knowledge of the application-module services. The user interface was implemented using IBM Presentation Manager, which requires a particular programming style. We expressed the design as specifications of Presentation-Manager

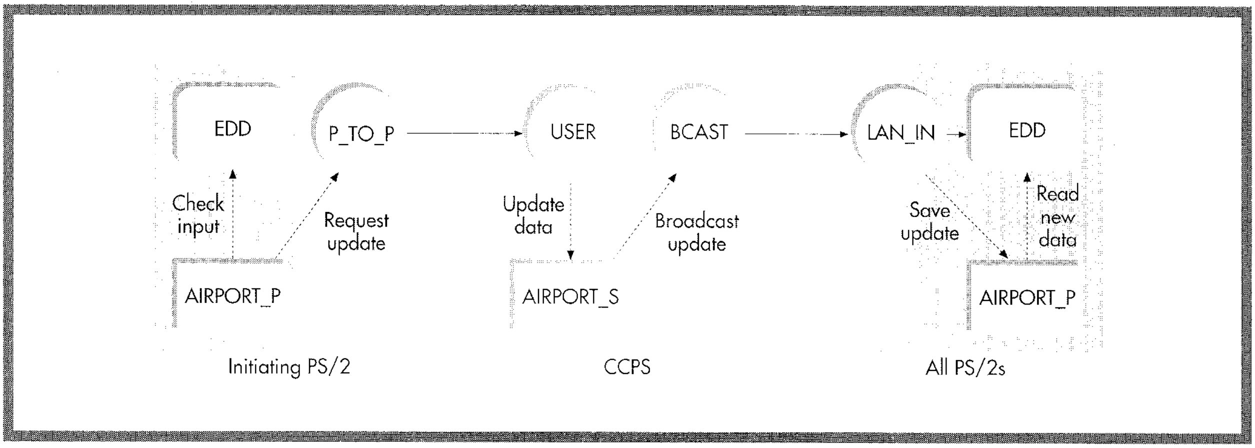


Figure 7. A typical CDIS transaction.

window classes, defining the messages and responses each could process.

Infrastructure design. The infrastructure mainly involved the LAN software. As part of the overall design, we developed a definition of the required LAN protocol and the interface between the LAN software and the rest of CDIS. We then produced a design for the LAN software itself.

The LAN software was a difficult area because the requirements were stringent: in-order delivery of all messages, without duplication, over two token rings with an automatic and invisible switch in case of failure. Performance considerations dictated that the unreliable broadcast service provided by the Netbios protocol had to be used as the basis of most communications, so a complex layer of CDIS software had to be built on top.

The requirements for the LAN software were initially expressed using VVSL. This was not entirely successful, because VVSL did not easily capture the protocol's dynamic behavior and the relationship between the behaviors at the two ends of the communication. For that purpose, we needed a notation that supported concurrency. We used the Calculus of Communicating Systems[8] for later parts of the requirements and for design. We translated the CCS processes directly into the implementation code. We used a form of value-passing CCS in which we incorporated VVSL definitions of data types and message structures.

The LAN software design was the only area of CDIS in which we used proofs. Because the LAN was such a critical component and its design so complex, we were concerned about its correctness. We attempted proofs of correctness in two areas. The first was at the interface between the LAN and the engineering part of the CDIS application. The LAN software detects failed machines and connections. It then either takes appropriate action or passes information to the rest of CDIS, which then makes decisions about stopping and starting failed machines. Because machines at each end of the link can fail at any time — even during restart after a previous failure — it is hard to eliminate the possibility of deadlock. To verify the protocol, we wrote a simplified version and used Edinburgh University's Concurrency Workbench[9] to establish temporal formulas representing facts such as "following a failure it is always eventually possible to recover." This gave us some confidence in the protocol's correctness.

The second area was recovering from message loss. We wrote two CCS processes: one representing a desired behavior specification and another intended to implement that behavior. We tried to prove that the implementation was correct, even when a loss was suffered during recovery. However, this was not possible because the protocol suffered from a subtle concurrency problem. Because it would have been almost impossible to find that bug during testing, attempting the proof let us correct the design rather than put a system into service that could at some point exhibit an incomprehensible fault. Details of this work have been published elsewhere.[10]

Findings. The use of formal methods in large-system design is less understood than their use in specification. We had some difficulties designing CDIS as formally as we would have liked.

Because design is many-dimensional, we were forced to use different methods for designing different parts of the system and there was no unifying way we could formalize the design as a whole — no theory of software architecture that we could use to validate the design. In particular, conventional refinement rules do not apply when the design structure differs from the specification structure — as it inevitably will in any large, distributed system. Our initial misunderstanding of this point led to problems at the interface between the application modules and the user interface, and a lot of necessary operations were missed in our first attempts to specify the application modules.

We had difficulty deciding on the right level of formality because the design was large and it was impractical to formalize all of it. We did not always make the right decisions. Formalizing too much gave us unnecessary work; in some pathological cases, the formal specification was more complicated than the code itself. Not formalizing enough meant that developers were sometimes faced with operations for which they had signatures but no clear definition of what was required. We probably should have adopted a simple rule: specify all update operations.

Despite these problems, the functional design was largely successful in

TABLE 2
EFFORT DISTRIBUTION ON CDIS IMPLEMENTATION

Activity	Days	% of Total
Requirements	270	2
Specification	1,274	8
Design	1,556	10
Code and unit test	5,219	34
Test and integration	2,458	16
Acceptance	723	5
User documentation	405	3
Project management	2,137	14
Development environment	549	4
Others	945	6
Total	15,536	

TABLE 3
EFFORT BY PHASE COMPARED WITH COCOMO

TPhase	CDIS	CoCoMo
Plans and requirements	13	7
Product design	14	17
Programming	45	47
Integration and test	28	29

generating a sound design that satisfied the specification. Few of the faults in CDIS can be attributed to an incorrect relationship between the specification and the functional design. The main benefit of the formal application design was that each module interface was precisely defined so clients knew exactly what they could rely on. We also found that, starting from the VVSL module specifications, the implementation of application modules was relatively straightforward and the code usually reflected the specification in a fairly direct way. This gave us good traceability between the code and the design. In some cases, the code was developed using rigorous correctness arguments.

As for infrastructure design, the CCS design of the LAN software was particularly successful in mastering the complexity of a difficult area and generating extremely reliable code.

We were unable to use a really formal process, such as carrying out proofs, to any large extent. This does not mean that such a process is always impractical; however, you have to balance the cost of using formality against the cost of *not* using it. From the developer's point of view, the purpose of a formal step such as a proof is to discover errors (there are other purposes, such as when a customer uses a proof for *assurance*, which has different costs and benefits). The expected cost of not carrying out a proof depends on the probability of an error existing that the proof alone would have discovered, along with the cost of that error. In the case of refinement proofs, we believed that the expected cost was small compared with the cost of actually doing the proofs; in the case of the LAN we believe it was large and easily justified our modest proof effort.

RESULTS

The operational CDIS software contains about 197,000 lines of nonblank, noncomment C code. A comparable amount of code was also written for test harnesses, emulators, and other nonoperational software. The specification documents were about 1,200 pages and the design documents about 3,000 pages.

As Table 2 shows, the total effort on the implementation project was about 15,500 person days. At about 13 lines of code per day, the overall effort was comparable with or better than other projects of the same size and kind (it is better than a simple Cocomo[11] prediction, for example). If anything, the use of formal methods saved rather than cost us effort. The distribution of effort is not very different from the standard Cocomo model, although as Table 3 shows, we did spend a slightly higher percentage of time on requirements and specification.

Faults. During development, developers delivered about 11 faults per thousand lines of code to the integration and system-test team. In the first 20 months after delivery, CDIS manifested about 150 faults — about 0.75 faults per KLOC. We believe this figure is significantly better than that on comparable projects.[12] Furthermore, few of these faults were specification or requirements problems, which often persist into the delivered system and prove costly to eliminate.

Customer's view. The Civil Aviation Authority was mainly affected by our use of formal methods in the system specification, rather than the design. From their perspective, the use of formal methods in system specification had both advantages and disadvantages. The main advantages were:

♦ The specification was comprehensive. The CAA team could refer to the specification to answer almost any question about what CDIS would do.

♦ The specification was precise. There was rarely any doubt about what it meant and how CDIS would behave.

♦ The use of the specification to derive the system tests meant that

CAA could see the level of testedness of CDIS at any time.

On the other hand, there were some drawbacks:

- It was difficult to get an overview from the formal specification, because it was neither top-down nor hierarchical.
- The core specification was a poor aid to internal communication at CAA. Because the notation was only understandable to the CAA team directly working on CDIS, they had to interpret the specification to other people in the organization.
- The user-interface specification was not as precise and complete as the core specification and thus was not as well defined as it should have been. Furthermore, the relationship between the two specifications was not always clear.
- The specifications were not clear about certain timing issues — in particular the point at which events became visible on the user interface.

The lessons that should be drawn from this for future use of formal methods are two. First, the formal specification should be accompanied by much more informal explanation using English text, diagrams, and any other convenient notation. Second, more emphasis should be placed on making the user-interface specification precise and complete.

CDIS was a large and complex project and many factors contributed to its success. One important factor was that we used formal methods with other good software-engineering practices. I think we have learned lessons about making that effective by combining formality with more comprehensible explanations and better specification structure.

Formal methods also contributed to our design process, but there are clearly many other aspects of good design. Carrying out effective formal design will require more work on understanding the different design dimensions and large-scale architectures.

Using formal methods helped us to build the right system and helped us to build it right — at no extra cost. Our project shows that using formal methods on real, large-scale projects is not only practicable but beneficial. The question software engineers should now be asking about formal methods is not *whether* to use them, but *how* best to benefit from them as part of a complete software-engineering approach. ◆

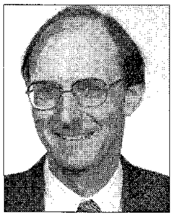
ACKNOWLEDGMENTS

This paper should really have as named authors all my colleagues in Praxis who worked on CDIS. It is their work that led to the results reported here. I hope that my opinions here do some justice to their efforts. I am also grateful to the CAA CDIS team, who supported our use of unfamiliar methods on a critical and difficult project.

I thank David Isaac who contributed the CAA's impressions of formal methods to an early version of this paper, and Glenn Bruns from the Laboratory for the Foundations of Computer Science at Edinburgh who helped us with the CCS proof work.

REFERENCES

1. D.J. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, 1988.
2. C.B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
3. N. Plat, J. van Katwijk, and K. Pronk, "A Case for Structured Analysis/Formal Design," *Proc. VDM '91, Lecture Notes in Computer Science No. 551*, Springer-Verlag, Berlin, 1991, pp. 81-105.
4. J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
5. C.A. Middleburg, "VVSL: A Language for Structured VDM Specifications," *Formal Aspects of Computing*, Jan.-Mar. 1989, pp. 115-135.
6. C.A.R. Hoare, *Communicating in Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
7. J. Jacob, "Refinement of Shared Systems," *The Theory and Practice of Refinement*, J.A. McDermid, ed., Butterworths, London, 1988, pp. 27-36.
8. R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
9. R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *ACM Trans. Programming Languages and Systems*, Jan. 1993, pp. 36-72.
10. G. Bruns, "Applying Process Refinement to a Safety-Relevant System," Tech. Report ECS-LFCS-94-287, Laboratory for the Foundations of Computer Science, University of Edinburgh, Mar. 1994.
11. B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
12. L. Hatton, "Programming Languages and Safety-Related Systems," *Proc. Safety-Critical Systems Symposium*, Springer-Verlag, New York, 1995, pp. 48-64.



Anthony Hall is a principal consultant with Praxis, where he led the analysis and design team on CDIS and has promoted the use of formal methods on many projects. His current interests are in formal aspects of software architectures and in tool-based verification of formal specifications and designs. He has published and lectured widely on formal methods, object orientation, and software engineering.

Hall is a chartered engineer. He received an MA and a DPhil in chemistry from Oxford and is a member of the ACM and the British Computer Society.

Address questions about this article to Hall at Praxis, 20 Manvers Street, Bath BA1 1PX, UK; fax: 44-1225-465205; jah@praxis.co.uk.