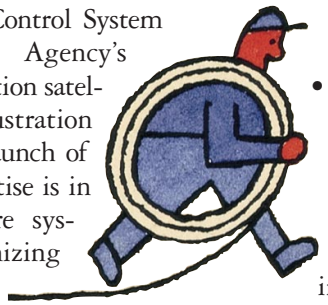
Daniel D. Corkill

Countdown to Success: Dynamic Objects, GBB, and RADARSAT-1

The most successful applications are never completed—they evolve with the enterprises they serve.

DYNAMIC OBJECT TECHNOLOGY ENABLES APPLICATIONS TO BE modified and enhanced during development and runtime without access to source code. The advantages of dynamic objects are often described abstractly, with general references to gains in developmental productivity and application flexibility. Dynamic objects play a pivotal role in Blackboard Technology's (BBTech's) blackboard and agent-based software products. In this article, I discuss the specifics of the important role dynamic objects play in BBTech's software products and in one of the applications built using them: the PLAN component of the Mission Control System for the Canadian Space Agency's RADARSAT-1 earth observation satellite (the background illustration appearing here depicts the launch of the satellite.) BBTech's expertise is in flexibly integrating software systems and effectively organizing and controlling the use of these systems in the resulting applications. There are three types of software systems that are integrated using our products:

- Legacy applications, sometimes written years ago
- Commercial software products, with integration and extension capabilities predetermined by their vendors
- Unimplemented systems being developed concurrently with the integrated application



For many customers, all three types of systems are involved.

Dynamic object capabilities greatly reduce the software-development effort resulting from the complexity and changing requirements of blackboard and agent-based applications. Our customers' applications often involve multiple generations of software systems spanning the entire organizational enterprise. The broad scope of these applications makes the ability to support change throughout their lifetimes even more crucial.

Our staff and customers use dynamic object capabilities at three distinct levels:

- In the creation, maintenance, and enhancement of our software products
- In the use of our software products by application developers
- In applications created using our software products

Dynamic Objects Supporting the Software

Generic Blackboard Builder (GBB) is an application development and delivery framework for high-performance blackboard-based applications. Blackboard-based processing is a powerful method for flexibly combining individually developed software systems and modules into a single integrated application [1–3].

The term "blackboard" is suggestive of the following scenario involving a group of people engaged in a brainstorming session: a group of specialists are in a room, with a large blackboard serving as the workplace for cooperatively developing a solution to the problem being examined. The session begins when the problem specifications are written onto the blackboard and a specialist makes the first contribution to the problem. Throughout the session, the specialists continue to look for opportunities to contribute to the developing solution. When someone writes something on the blackboard that allows other specialists to apply their expertise, those specialists record their contributions on the blackboard, enabling additional specialists to apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved.

A blackboard application achieves this flexible "brainstorming" style of interaction using computer programs as the specialists. A blackboard application consists of three major components (Figure 1):

- Software specialist modules, called knowledge sources (KSs). Like the human experts, each KS provides specific expertise needed by the application.
- The blackboard, which is a shared repository of problems, goals, partial solutions, suggestions, and contributed information. The blackboard can be viewed as a dynamic "library" of requests and contributions that have been recently "published" by other KSs.
- The control shell, which strategically controls the flow of activity in the application. Just as eager human specialists sometimes need a moderator to prevent them from trampling one another attempting to grab the chalk, KSs require a mechanism to organize their use in the most effective and coherent fashion. The control shell provides the machinery for managing KS computations in an application.

allow us to provide software updates and bug fixes that can be easily sent by email or downloaded from the Internet.

THE GBB PRODUCT IS BASED ON FIVE YEARS OF academic research concerning the software technology necessary to support blackboard-application development. Development of the commercial GBB product required availability of standardized dynamic-object capabilities in order to lower development costs and ensure wide availability of the completed product. Development cost was an important business consideration at the time due to the general economic recession and the near collapse of the artificial intelligence marketplace, with which GBB would be associated. Successful commercialization of the academic research that preceded GBB would require rapid, low-cost development of the commercial product. Finally, in 1989, progress toward ANSI standardization of Common Lisp and the Common Lisp Object System (CLOS) enabled GBB to be developed, maintained, and enhanced by a small technical staff.

One of the important lessons we learned from the academic research was that a single-inheritance approach to blackboard objects was unwieldy, due to the need to combine object-modeling classes with

control capability and other classes. For the GBB product, we sought to provide a library of blackboard-object capabilities that could be easily combined to develop application objects. These object "mixins," multiply inherited independent classes, provided basic blackboard-object properties and allowed different control-shell facilities to be transparently added to blackboard objects.

Another advantage provided by our use of CLOS is its extensible nature, which allowed us to implement the GBB "language" without redeveloping the extensive language capabilities already present in CLOS. The result is that GBB retains all the richness of CLOS, thereby providing a well-developed product language to GBB application developers.

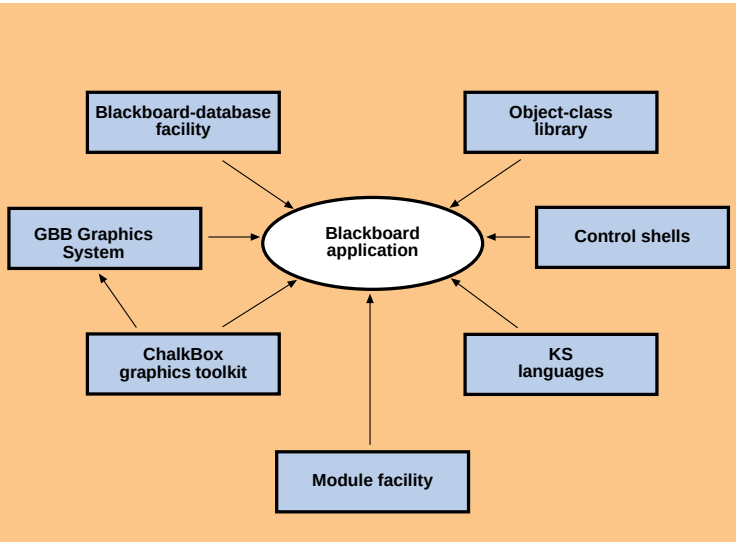


Figure 2. GBB's major subsystems

In terms of product maintenance and enhancement, dynamic objects allow us to provide software updates and bug fixes that can be easily sent by email or downloaded from the Internet. Even if these updates involve changes to the GBB's internal classes, they can be automatically loaded and incorporated into the user's environment.

Finally, the ANSI CLOS standardization effectively eliminates many hardware-platform and software-vendor dependencies, allowing us to support GBB in a wide range of configurations (with low support and maintenance costs).

Dynamic Objects in the Use of GBB

GBB is not a monolithic tool. It consists of a number of major subsystems that can be used during application development and runtime execution (Figure 2). These components include: the blackboard infrastructure, control shells, KS languages, monitoring facilities, graphics, utilities, and blackboard-object mixin libraries.

Developers and users can mix and match GBB components as needed. For example, a developer or user may decide to use GBB's interactive blackboard browser to view a portion of the blackboard. GBB can dynamically load the ChalkBox and GBB Graphics System components into the running application and begin to browse the blackboard.

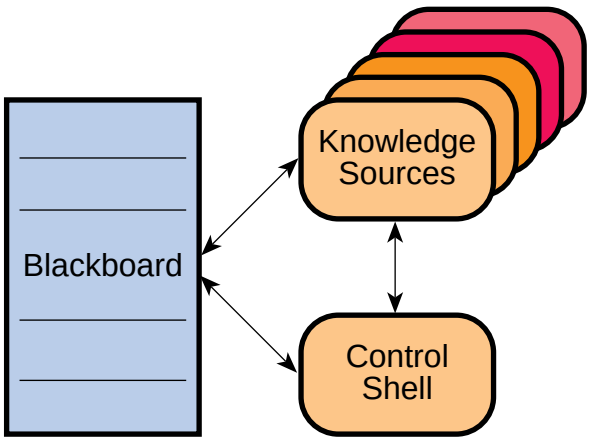


Figure 1. Major blackboard application components

One BBTech customer was giving a demonstration of an application for analyzing electronic devices. Part of this application included a specialized graphical user interface that had been developed for presenting analysis results. One of the visiting senior managers was impressed with the application but wanted to see the distribution of candidate designs over two attributes that were not supported by the GUI tool. The implementers mentioned that their GUI did not currently have that capability, but the underlying GBB system had a blackboard-browsing tool that could display the design objects along a number of dimensions, including the ones in question. They loaded the GBB Graphics System, and in a matter of seconds the requested results were visible. Such an immediate response to unplanned requirements would not have been possible without dynamic objects and the flexibility of GBB.

GBB also uses dynamic objects to increase efficiency. Consider the situation of adding a new KS to the application. In GBB, KSs are triggered in response to events that are of interest to the KS. Events include the creation and deletion of blackboard objects; initialization, accessing, and updating of object slots; and linking and unlinking of objects—as well as software-signaled and asynchronous external events. A particu-

lar application has thousands of possible events, with only a few hundred of interest to KSs at a given time.

For top performance, only those methods involving events that are of interest should have the overhead of event-based control attached to them. GBB dynamically recompiles event-related methods when KSs are added to or deleted from the application. Without the ability to redefine methods on the fly, GBB applications would have to be entirely recompiled whenever the events of interest change, or they would incur the overhead of unused event signaling throughout.

Dynamic Objects in GBB Applications

GBB was designed to deliver all the dynamic object capabilities of CLOS to developed applications. These capabilities include object-based abstraction and encapsulation, multiple inheritance, and a choice of dynamic or static typing. GBB provides a smooth extension of Common Lisp and CLOS, with blackboard-specific language capabilities augmenting the base language. In a practical sense, the GBB "language" encompasses ANSI Common Lisp and CLOS.

Dynamic object capabilities are especially important in blackboard-based applications for the following reasons:

- Blackboard objects provide the basis for developing a rich and highly specialized representation language for the information needed by the individual software systems. Dynamic, multiple inheritance allows developers to augment generic object classes with specialized application information.
- Individual KSs should not manage the storage of blackboard objects. There is no way a KS writer can—or should—predict when other KSs can use blackboard objects. Automatic storage management decouples object management responsibility from individual KSs.
- Each individual KS should be required to know only the details of the portions of blackboard objects that relate to it. Changes to blackboard-object classes and methods that are not directly relevant to a KS should not require its recompilation. The ability to change classes at runtime is important to long-lived applications. In continuous-availability applications, it is particularly useful if the individual objects of a changed class are converted lazily; that is, as each object is accessed.
- The application should change behavior according to the current mix of KSs. It should be possible to add, remove, or update the application's KSs dynamically. Dynamic loading and the creation and modification of object classes and methods provide the flexibility to easily change the set of KSs.
- Control and coordination of KSs is separated from the individual KSs. KSs do not call one another directly; control decisions are initiated as a result of operations on blackboard objects and other processing and external events. Again, the use of dynamic objects provides the flexibility to easily change the control decisions and mechanisms without requiring changes to the KSs.
- GBB views KSs as "black boxes." They can be written in a variety of programming languages and can be executed locally or on remote machines. In some cases, a KS can be an interaction with a user. Dynamic objects are used to present a uniform KS interface that allows the implementation of individual KSs to be changed easily—by changing the classes of KS objects or the code used to implement them. Dynamic-KS capability allows new versions of KSs and additional KSs providing new capabilities to be added to running applications.

Categories of Applications

THERE ARE TWO BROAD CATEGORIES OF APPLICATIONS built with GBB: *collaborative-integration applications*, where closely cooperating software systems are integrated into a collaborative enterprise; and *complex problem-solving applications*, where search and other knowledge-based techniques are applied to a complex problem.

Application areas in each category span a broad range: from engineering design to financial analysis, sensing to process control, and simulation to planning and scheduling. Although the specifics differ in each application, each gains the advantages of dynamic objects.

In the remainder of this article, I will take an informal look at the important role dynamic objects and GBB played in a specific GBB application: the RADARSAT-1 Mission Control System.

A Case Study: RADARSAT-1

RADARSAT-1 is a versatile earth-observation satellite that was launched on November 4, 1995. RADARSAT-1 is the first in a series of synthetic aperture radar (SAR) remote-sensing satellites designed to allow end users anywhere in the world to connect to the system, prepare and submit requests, and monitor the status and progress of each request as it is planned and executed. Unlike optical-imaging systems, RADARSAT-1's SAR instruments send pulsed microwave signals to earth and then process the pulses that are reflected back to them. This

enables RADARSAT-1 to collect data under all atmospheric conditions—clouds, rain, dust, or haze—by day or night. (See http://www.rsi.ca for more information.)

The space-based portion of the RADARSAT-1 system consists of a satellite with its SAR illuminator, sensors, an onboard recorder, and communication facilities. The satellite is in a near-polar orbit and provides complete earth coverage in 24-day cycles. The satellite has a phased-array antenna, allowing users to choose from more than 20 different beams and instance angles for imaging. The ground-based portion of the system includes the data-receiving stations located around the world and the data-processing operations required to transform the raw digital data into images.

The RADARSAT-1 Mission Management Office System

At the heart of the RADARSAT-1 operation is the ground-based Mission Management Office (MMO) portion of the Mission Control System. The MMO system is responsible for translating requests from commercial and government users into detailed schedules of coordinated activities for the spacecraft payload and the ground reception and processing facilities. There is no single solution for every combination of user requests and system resources: a good schedule may be achieved through many possible combinations and activity sequences. The MMO system also maintains the health and operation of the spacecraft.

The MMO system must produce and maintain a schedule of activities for the spacecraft payload and ground systems: the radar transmitter and receiver on the spacecraft; the tape recorder on the spacecraft; the downlink transmitter; data-reception stations; and data-processing facilities. All of these activities must be planned and coordinated for time segments as small as fractions of a second. The satellite orbits the earth every 100 minutes, and a typical imaging activity that covers 100 kilometers on the ground may last only 15 seconds. There may be many users requesting imaging for different segments of each orbit.

The RADARSAT-1 system is sponsored by a number of participating partners,[1] and there are agreements between the partners that allocate the sensor utilization among them. Thus, in addition to the physical constraints of the spacecraft, data-reception stations, and data-processing facilities, the MMO system must accommodate access policies and priorities among the participating partners and commercial customers. The MMO system must enforce these policies and also allow for changes to the policies on an ongoing basis during the operational life of the system. In fact, most of the policies were not determined until after the MMO system had been placed into service.

The MMO system must model each of the system elements to determine whether proposed activities violate either a system constraint or a policy constraint. For instance, because of power limitations, the radar transmitter can only operate in imaging mode for a maximum of 28 minutes for each orbit period. If the MMO system allocates too many imaging activities for a given period of time, this SAR on-time constraint would be violated. The system has to detect these constraint violations and resolve the conflict before it can issue the activity schedules.

of users and requests, it is impossible for a single planner to cope with the planning load.

At the heart of the MMO system is the Systems Operations Plan (SOP), a continuous time-based model of the past, present, and future of the mission. As time progresses, near-term sections of the SOP are frozen and executed by the spacecraft, ground stations, and processing facilities.

Part of executing the SOP is generating the payload command data and passing it to the Mission Control System. The MMO system also sends off reception schedules to the different ground stations and detailed processing information to the image-processing facilities.

There are over 140 system constraints that define the limiting conditions of the various system resources involved in imaging, recording, transmitting, and processing the image product for RADARSAT-1. The large number of constraints makes planning activities a very complex problem.

[1]The partners include: the Canadian Space Agency (CSA) through the Canadian federal government; the Canadian Provinces of Saskatchewan, British Columbia, Quebec, and Ontario; NASA; NOAA; and RADARSAT International Corporation (RSI). Industrial partners include: Spar Aerospace (Montreal); Ball Aerospace, Space Systems Division; McDonnell-Douglas; MacDonald Dettwiler & Associates; CAL Corporation; COM DEV; Fleet Industries; IMP and FRE Composites; DORNIER; and ODETICS.

Like most planning systems, the MMO has no single correct answer or optimum solution for every combination of user requests and system resources.

The mission life for RADARSAT-1 is five years. It is anticipated that there will be at least two additional RADARSAT spacecraft launched to continue the mission. Images taken early in the RADARSAT program must remain accessible to the users throughout the mission life, and users of the system can ask for data from a catalog of previous image strips, as well as for newly acquired data. This means that the mission-planning process must span the entire life of the mission; it must operate with a large database and a large number of images.

Given the number of users and user requests, it is impossible for a single planner to cope with the planning load. Multiple planners must be able to work simultaneously without getting in one another's way. Concurrent planning is complex and is complicated further by working with relatively long transactions.

MMO PLAN Software Development

The Canadian Space Agency (CSA) assigned responsibility for developing the MMO software for RADARSAT-1 to MacDonald Dettwiler & Associates (MDA) of Richmond, British Columbia (see Table 1). Initially, MDA conducted approximately one year's worth of software-engineering work for the MMO system, beginning in the summer of 1991. This work led to wildly divergent views on what the MMO system requirements ought to be, what technologies were needed, and how much the development should cost. They conducted a structured analysis, based on a top-down functional decomposition and hardwired scheduling capabilities, that led to an estimated development cost of approximately $13 million for

A NEW SCHEDULING EFFORT

Two members of the MDA development team, Peter Macdonald and Bruce Foulger, are now using dynamic objects and GBB to develop a new line of airline-crew scheduling products at Mercury Scheduling Systems in Vancouver, Canada. Crew costs account for an increasingly significant portion of airline operating expenses. Effective scheduling of crew resources enables airlines to reduce operational costs and increase efficiency.

As with the RADARSAT PLAN component, airline-crew scheduling involves numerous policies and constraints; and adaptability of the crew-scheduling software is even more important than with RADARSAT. Each member of the cockpit crew must be qualified to fly the specific aircraft configuration. Long flights require an in-flight relief crew. Flight attendants must be qualified for the aircraft type and their position within the cabin. Language requirements and differences must be considered. Each crew member needs specific amounts of rest time between flights, with global time-zone changes fully accounted for. Lastly, government regulations, union industrial rules, and the commercial objective of the airline itself must be considered.

All of these policies and constraints are subject to change. For example, airlines in Europe are currently preparing for a new set of European safety rules governing the duration of on-duty periods.

Mercury has been providing innovative airline-crew-scheduling services for the past 14 years. Termed preferential bidding, Mercury's scheduling approach lets crew members specify detailed personal preferences. These preferences include flight dates, destinations, aircraft, and other crew members. The preferences are combined with crew qualifications and training time to produce schedules that accommodate crew preferences to the greatest extent possible.

The primary market for Mercury's new crew-scheduling products is the nearly 200 medium-scale airlines worldwide. Unlike the major carriers, who generally custom-develop their scheduling software, these airlines are seeking off-the-shelf scheduling software that can be economically customized to meet their specific scheduling needs. Additionally hundreds of small-scale carriers that currently handle scheduling manually may now be able to justify the cost of Mercury's new software products.

Mercury's basic approach is to develop a core scheduling product that can be customized to the specific needs of each airline. Mercury not only sells scheduling software products, but also configures them to meet the different policy needs of different airlines, as well as the changing policy needs of a single airline. The flexibility provided by dynamic objects and GBB will allow Mercury to work very closely with their customers in tailoring the scheduling products to individual customers. Mercury's approach will allow an airline to pay substantially less for an off-the-shelf product that can still be customized to their needs. Only the parts of the scheduling problem that are really unique to an airline will be custom-developed.

the MMO Plan software.

The CSA had initially budgeted $4 million for the MMO system. They were also concerned about whether the MMO system should be more of a planning system or a scheduling system and how much "flexibility" either approach would provide. MDA was concerned about the difficulties in specifying such things as "flexibility" and about the time required to build complex systems like the MMO from scratch. The situation was tense, as there were only 2½ years to develop the mission-critical MMO software. The spacecraft was scheduled for launch in 1995, and the MMO software needed to be completed and validated well in advance of the launch.

1989	RADARSAT International (RSI) established
Summer 1991–Summer 1992	Initial MDA systems-engineering work
July 1992	PLAN system development started
December 1992–January 1993	Extensive survey of commercial tools conducted
February 15–19, 1993	CSA and MDA conduct preliminary GBB study at Blackboard Technology
March 1993	MDA begins using dynamic objects and GBB
August 1993	Software transferred from SunOS to Solaris 2.0
Fall 1993	"Mini slice" completed
Fall 1993–January 1994	End-use-scenario work
Spring 1994	Look-and-feel development
Spring 1994	OBR removed for cost reduction (later added back in)
May 1994	PLAN, "performance slice" coding proceeds in earnest
August 1994	Context mixins added
September 1994	Major development demonstration achieved
May 1995	MMO software formally accepted by CSA in Ottawa
July 31, 1995	RADARSAT shipped from Ottawa to Vandenberg Air Force Base
August 28, 1995	Mission Control System-to-spacecraft compatibility test completed
November 4, 1995	RADARSAT-1 launched
November 28, 1995	First RADARSAT-1 image taken
December 11, 1995	Final orbital position attained
December 14, 1995	First RADARSAT-1 image publicly released

Table 1. RADARSAT-1 timeline

PLAN is the central and largest component of the MMO system. Determining the requirements for PLAN was a bit difficult, since RADARSAT-1 was the first commercial SAR satellite developed and there was very little national or international expertise to draw upon. This led to one of the major system requirements for PLAN, which was to develop an extensible system because the CSA was certain there would be major enhancements and modifications as they learned how the system would be used.

In July 1992 a Working Group was established at CSA consisting of engineers from SPAR, CSA, and MDA. The Working Group was given two tasks: to develop a common understanding of what was required and how best to achieve it, and to develop a project plan to meet schedule and budget constraints. There was a sense that perhaps some rapid prototyping might help to build understanding. The group decided to bring new engineers into the Working Group and start again, without any preconceptions.

Given the complexity of the planning task, the Working Group made the following decisions:

Use object-oriented analysis (OOA) rather than functional decomposition to formulate the problem. OOA provided a way of handling the considerable complexity of the MMO system while maintaining a fairly direct relationship with implementation and cost issues. It also facilitated a layered design and provided a clean separation between unchanging, fundamental components of the system, such as earth-satellite beam geometry, and less understood aspects, such as sensor contention-resolution strategies.

Use a blackboard architecture to support the required incremental planning activities and to provide flexible, opportunistic control. The goal was to design a system that was "automatable" rather than automatic, since it was impossible to predict the future policies that would need to be provided to whatever scheduling architecture was adopted. By taking a blackboard approach, the scope of initial development could be reduced while maintaining the ability to add practical enhancements in the future.

Use a risk-driven spiral-development process, rather than the classic waterfall model. The Working Group acknowledged that there were aspects of the MMO that were impossible to solve without building a model first. There was a concern that spiral development might be misconstrued as "code first, think later," and the Working Group believed that there were many aspects of MMO that could be formulated through analysis rather than code writing. Although one of the original ideas behind the Working Group was to do rapid prototyping, a bottleneck in procuring prototyping tools prevented this. In the view of Peter Macdonald, Project Architect for MDA, "This was a fortunate turn of events. What was needed at this stage was a more suitable formulation of the problem and design approach, rather than rapid prototyping."

After five months of work, the Working Group had made substantial progress including:

Generation of an initial OOA problem formulation, captured in the form of a black-box level-requirements specification. The specification identified specific requirements for "adaptability" in place of earlier requirements for such features as automatic scheduling. In contrast with the waterfall software-development approach, the requirements specification used during the spiral-development process was iteratively and incrementally revised during MMO-system development. As a result, it became an invaluable reference of the application problems that needed to be solved.

Generation of an initial high-level design. This design included a process-level decomposition and a specification of the domain-independent layers of software needed to support GUIs, persistence, and multiprocess synchronization and messaging. The high-level design also described how a blackboard architecture could be used for the PLAN component, and how this approach directly related to the "adaptability" issues outlined in the requirements.

Completion of a new project plan. The projected cost of the MMO was now $8M, which was substantially lower than the earlier $13M projection. Much of the cost reduction was due to the removal of functionality such as automatic scheduling and to savings achieved through better design. Unlike the initial work, flexibility and adaptability were to be achieved through design, rather than through an increase in development effort.

Identification of major risks. The risks identified at this stage included such items as database performance, user interface look and feel, and some areas of the problem domain that were not well understood. Another important risk area involved the proposed use of a blackboard architecture for PLAN. According to Peter Macdonald, "When a blackboard architecture was first proposed for PLAN, we were unaware of GBB. We didn't know if we would be able to find a commercial framework that could handle the size and complexity of our problem and still provide the necessary performance without introducing a myriad of integration headaches. To guard against the possibility of not finding such a tool, we included estimates in the project plan for development of the necessary infrastructure by MDA and recommended that every effort be made to find a suitable commercial tool." The Working Group was given an additional four-month mandate to mitigate the largest of these risks.

In retrospect, MDA's decision to use a blackboard architecture for PLAN was a sound one, given the complexity and flexibility requirements of the system. Initial plans for developing the required infrastructure, if necessary, included using:

- C++ to represent SOP objects, constraints, and policies
- Commercial C++ libraries (Booch components) to reduce the development effort in creating SOP objects
- A commercial C++ development environment (Object Center) for exploratory development

HOWEVER, IT QUICKLY BECAME APPARENT that dynamic, high-productivity software-development tools would be needed to develop PLAN on time and within budget. According to CSA's Larry Cochran, "We first looked at a build-from-scratch approach and saw that we simply didn't have enough time or money to invent a solution to the generic planning problem and then make it work for RADARSAT. We felt that there must be some tools out there that would provide a framework for developing the mission-planning program."

Early in 1993, MDA contacted BBTech regarding the feasibility of using GBB to develop the PLAN system. Concerns centered around the following questions:

- Did GBB, a CLOS-based artificial-intelligence technology, provide the performance needed for the PLAN system?
- How easily could systems written in other languages (C and C++) be integrated with GBB?
- How much of the previous development work on defining an object model could be directly transferred into GBB?
- How quickly could a highly competent programming staff learn to use GBB and CLOS?

MDA developed a set of benchmarks for: (1) evaluating how well GBB would perform as the basis for PLAN, (2) a six-week "mini slice" development effort to integrate the GBB-based PLAN system with all of the operating systems, compilers, and GUI tools that would be used, and (3) a six-month "performance slice" effort to identify any performance issues in loading temporal portions of the SOP from the Sybase repository and to address issues in incremental database updates during PLAN activities.

With these evaluation plans in place, the MDA team[2] and CSA's Larry Cochran spent a week evaluat-

ing GBB at BBTech's offices in February 1993. MDA and CSA liked what they saw in GBB and made a good start toward completing the six-week mini slice during the week. By March 1993, MDA and CSA were convinced that GBB was a robust commercial tool that would save substantial development effort while providing very good performance. According to Peter Macdonald, "We were very impressed with GBB's clean design and the product's robustness, as well as the excellent level of support provided by BBTech. We also had a good idea by then of how to integrate GBB with our other layers of design, and we were very comfortable with the 'openness' of the GBB framework and CLOS."

After choosing GBB, the next step involved completing the mini slice of the system. This was a prototyping exercise to integrate GBB with all of the operating systems, compilers, and GUI tools that would be used in the MMO environment. According to Larry Cochran, "It's not an easy thing to get a wide range of tools, compilers, operating systems, and software packages to work together. We found a few glitches in our exercise, but CLOS and GBB provided a good vehicle for debugging those interfaces." Development of the mini slice removed critical integration risks early on, before the bulk of the code had been written. One outcome of this work was a decision to move from SunOS to Solaris 2.0.

In April 1993, MDA revised the overall MMO project plan given a GBB-based PLAN component. The savings arising from the decision to use GBB resulted in a revised projected cost of $6 million. SPAR, CSA, and MDA negotiated a new contract for the balance of the MMO development based on the newly established baseline.

From the fall of 1993 through January 1994, extensive end-use-scenario work was performed for PLAN. Emphasis was placed on logical-level problem-solving scenarios that helped both to refine the information model and to uncover the choice points in manual problem solving, the views required, and the behavior of important planning-control objects. This end-use-scenario work was done in conjunction with several of the CSA personnel who were to be responsible for the actual planning once the mission was under way.

This work progressed well but uncovered two areas of the problem domain that were more complex than originally anticipated. The most critical area was the control and utilization of the onboard recorder (OBR). Detailed modeling of tape acceleration and deceleration times was needed to keep track of which SAR data are stored on which part of the tape at any point in time. Propagation of time-measurement uncertainties was needed in order to derive pessimistic models of which part of the tape needed to be played back to a ground station and how much overlap was needed between ground-station receptions so that portions of imagery could be recombined on the ground.

The second important area dealt with the interactions among multiple, concurrent planners.

DURING THE SPRING OF 1994, MDA EXPLORED look-and-feel development of PLAN. The separation of logical-level problem-solving scenarios from the look-and-feel development was very successful. The logical-level scenario work enabled agreement to be reached on problem-domain issues, which then formed a stable basis for addressing visualization and fine-grained user-interface-level dynamics. Once the logical scenario work was complete, consensus on look-and-feel issues was achieved quickly.

Also during the spring of 1994, MDA revised its project plan and estimates to completion. Increased complexity in several key areas increased the projected cost, and MDA reduced the scope of development in a number of areas, including removal of OBR support, to bring the projected cost back to the original budget. The modularity of the blackboard-based design, the natural mapping between the PLAN problem domain, blackboard objects, and KSs, and the flexibility provided by dynamic objects made it easy to scale back development in these areas.[3]

In May 1994, work on the performance slice started in earnest. In addition to working on PLAN, the MDA team completed interfaces to the other portions of MMO, such as the long-term Sybase image repositories and the Order Desk facility. The performance slice was successfully demonstrated in September 1994.

One example of the benefits of using dynamic object capabilities in the development of PLAN was the addition of a context-transaction mechanism midway through the project. PLAN's constraint-checking component was separate from the action operators generated from user requests, making it easy to add, remove, and modify constraints without changing the code used to implement the core actions.

When planned actions are added in response to a request, KSs for checking constraints are triggered. When constraints are violated, planned actions must be removed. MDA decided that a context-based trans-

[2]The MMO PLAN development team consisted of: Peter Macdonald, Project Architect for MDA; Don Smith, Project Manager; Judith Vosko, Project Engineer; Bruce Foulger; Gail Godbeer; Liya Guo; and Rod McGill.

[3]The OBR support was later reintroduced.

action mechanism was the best way to undo planned actions.

For performance reasons, GBB does not include a built-in transaction mechanism for all blackboard objects. However, a specialized context-mixin class was supplied by BBTech's Kevin Gallagher. Consisting of about 100 lines of code, the context-mixin class provides objects with a current context and any number of previous contexts maintained as a stack. Including this mixin class in the parent classes of a blackboard-object class adds transactional context behavior to the normal behavior of all slot accessors. Dynamic object capabilities allowed this mixin class to be added to any of the already defined classes that required transaction capability. It did not require any knowledge of the structure or even the identity of the classes into which it was being mixed. Finally, it significantly reduced the amount of coding required.

An example of the use of dynamic object flexibility and GBB's extensibility is the late reintroduction of the satellite's OBR to the PLAN system. Scheduling the use of the OBR was eliminated from PLAN control due to financial constraints. However, once it was determined that most of the images were to be recorded onboard, OBR scheduling was added back in—after the system was delivered. The OBR was by far the most complex satellite resource to deal with at the software level, yet there was no problem updating the imaging model, adding new actions, updating the views, and adding new constraints.

The Results

The decision to use dynamic objects and GBB resulted in the completion of the software on schedule and at substantial cost savings. The marriage of dynamic object capabilities with the blackboard facilities provided by GBB proved effective during PLAN development and during the RADARSAT-1 mission.

According to Larry Cochran, "We found dramatic cost benefits for the program by using dynamic objects and GBB. Our costs dropped to about one-third of our original estimates."

"The system works very well," explained Mohamad Farhat, Software Systems Manager for the mission. "It pulls in requests from around the world, 24 hours a day, and we've never missed a product [delivery]."

Flexibility and extensibility are vital to the mission's success, according to Ken Lord, Project Manager for the operational phase of the mission. "The MMO's database has controllers with blocks of software that generate the products that interface with all functional areas. You can easily build the system in modules. For example, there are plans now to bring stations on in Singapore, Japan, Australia, and Brazil. Each time we bring on a new station, we just add a new set of controllers to interface with the receiving station for the download of image data."

Summary

Blackboard applications, such as those built with GBB, benefit greatly from dynamic object capabilities. The most successful blackboard applications are never truly "completed." They continue to grow and evolve in tandem with the enterprises they serve. The RADARSAT-1 experience is but one example of the successful use of dynamic objects and GBB (see the sidebar "A New Scheduling Effort").

GBB itself would probably not exist without dynamic objects. A more static product would have been far more expensive to develop and more difficult to use, and the resulting applications would be less flexible and adaptable to change. GBB's use of dynamic objects gives GBB-based applications the flexibility needed to operate effectively in fast-paced, open, dynamic environments. C

Acknowledgments

The author would like to thank Peter Macdonald and Bruce Foulger for sharing their recollections of the RADARSAT-1 development effort. The views expressed in this article are those of the author and do not necessarily reflect those of the Canadian Space Agency, MacDonald Dettwiler & Associates, or RADARSAT International Corporation. Zack Rubinstein and Suzanne Tromara provided useful comments on early versions of the article.

References

1. Corkill, D.D. Blackboard systems. *AI Expert* 6, 9 (Sept. 1991), 40–47.
2. Engelmore, R.S. and Morgan, A., Eds. *Blackboard Systems*. Addison-Wesley, Reading, Mass., 1988.
3. Jagannathan, V., Dodhiawala, R. and Baum, L.S., Eds. *Blackboard Architectures and Applications*. Academic Press, 1989.

Daniel D. Corkill (corkill@bbtech.com) is President of Blackboard Technology (www.bbtech.com) in Amherst, Mass.