Planning and Scheduling for Regional Validation Centers

Robert F. Cromp
Code 935
NASA/Goddard Space Flight Center
Greenbelt, MD 20771
cromp@sauquoit.gsfc.nasa.gov

John R. Bane
Global Science and Technology, Inc.
6411 Ivy Lane, Suite 300
Greenbelt, MD 20770
bane@gsti.com

Abstract

Early in 1997, Code 935 at NASA/Goddard Space Flight Center installed the prototype version of its Regional Validation Center (RVC) system at four university sites. The RVC system is a framework for acquiring, processing, indexing, storing and retrieving satellite image data. This paper describes the planning and scheduling components of the RVC, which are responsible for sequencing algorithms to produce products and for assigning computational resources to those sequences.

Introduction

Over the past decade, computer science researchers in the Applied Information Sciences Branch, Code 935, NASA/Goddard Space Flight Center have developed an end-to-end scientific spatial database management system called the Intelligent Information Fusion System (IIFS) (Cromp, Campbell, & Short 1993)(Short *et al.* 1995) with the express purpose of developing, incorporating and evaluating state-of-the-art techniques for handling EOS-era scientific data challenges. Since 1989, the IIFS has been based on an object-oriented database which is used to store metadata about large scale data holdings. The metadata itself is organized to enable fast, efficient access to the appropriate data sets. To handle image data, our research group has developed an innovative spatial data structure known as a sphere quadtree (SQT) that more naturally represents data acquired globally. Additionally, we have developed a number of fast techniques for automatically extracting information about image content, enabling users to query for pertinent data sets based on the features of scientific interest within the images themselves. The IIFS includes a planner/scheduler/dispatcher module (the PSD) that monitors and assigns system resources for processing the data flow, including all processing of higher level products, and extraction and assembling of metadata.

The IIFS is a domain-independent search engine for managing large volumes of data. We have recently embedded the IIFS in an end-to-end remote sensing system which acquires satellite data as the satellite passes from horizon to horizon, and gives users the capability to query for data products using a graphical interface. The resulting system is known as a Regional Validation Center (RVC).

Earlier this year, Code 935 at NASA/GSFC installed a prototype version of the RVC system at four university sites (University of Maryland/Baltimore County, Clemson University, University of Southwestern Louisiana, and University of Hawaii). Each site acts independently as its own RVC, and customizes the system to match their own remote sensing applications.

The RVC system is intended to start out as a small, low-cost way to capture, process, index, store, and retrieve satellite image data, with the potential to grow in capability as its data store and the needs of its users grow. The initial configuration at our prototype sites consists of a COTS satellite dish with an associated PC-class control computer, a medium-power Unix workstation to process the data and store the metadata, and a small robot tape drive to store the data and products. If an RVC needs more storage or processing power, it can simply add more Unix workstations as resources (see below).

Users at an RVC organize their system around a central theme, customizing the RVC software by developing and registering specific algorithms which serve to organize and enable retrieval of the RVC's data holdings. Given the finite availability of computing resources, and the complexity of product chaining permissible in an RVC, it is essential that the appropriate planning and scheduling occur to satisfy the users in a timely manner.

The planner/scheduler must be able to respond in a goal-directed mode to handle users' queries, and in a data-driven mode whenever new data is ingested from a satellite pass. We present our approach at performing these tasks in the remainder of the paper.

The RVC Framework

System Components Figure 1 presents the components of the RVC and the general information flow between them.

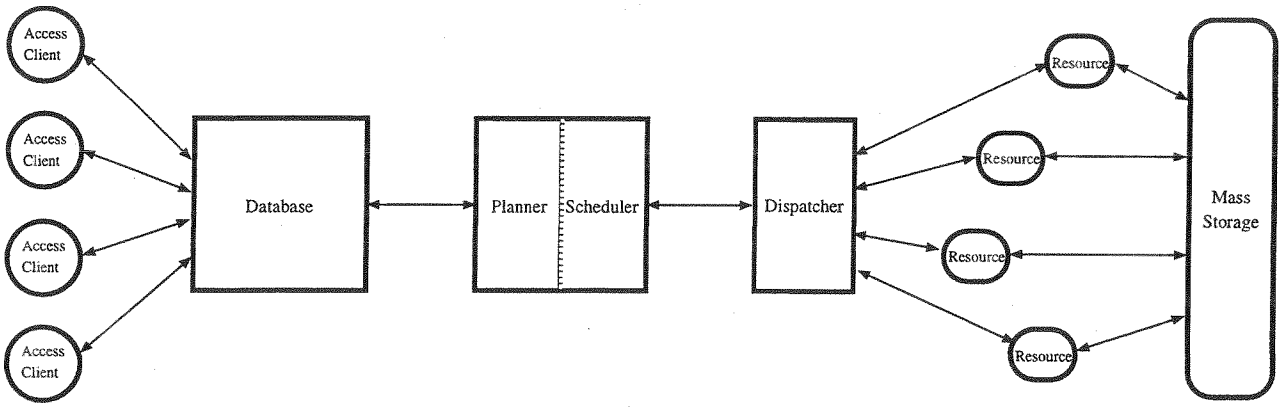

Figure 1: A simplified Block diagram of the components of the RVC system. Each block in the diagram represents a separate process; the RVC is a peer-to-peer system that coordinates the efforts of its parts by passing information via network sockets.

The components function as follows:

Access Clients - connect to the Database to submit queries and product requests. Currently we have implemented one access client for users called `rvci` and two maintenance/curator clients called `algo` and `resource`. The current access clients are written in Tcl/Tk.

Database - an object-oriented database holding the metadata for observations as well as descriptions of compute resources and programs the RVC can use to generate products. Actual image data is not stored in the database; it lives in the Mass Store and is referred to in the database via URL. The current RVC database is a C++ application using ObjectStore.

Planner - a system that accepts product requests and generates sequences of algorithms that will fulfill the requests. Product requests come in as lists of goals (types of products) and initial conditions (which observations to use and what products derived from those observations are available). Plans are passed to the scheduler as directed acyclic graphs of algorithms linked together by temporary files representing the inputs and outputs of the programs. The current RVC planner is Simple Non-Linear Planner (SNLP) (McAllester & Rosenblitt 1991), chosen primarily because it is small and hence easily modified. SNLP is written in Lisp.

Scheduler - a system that accepts plans and binds resources (CPU cycles and disk space) to them so they can be executed. The scheduler maintains a resource availability database to help it do its job. The current RVC scheduler is descended from a prototype scheduler named Data Archiving and Distribution System (DADS) (White *et al.* 1996) done for EOSDIS by Honeywell; it models the temporal availability of resources using their Time Map Manager (TMM) temporal constraint system. TMM and DADS are written in Lisp.

Dispatcher - a system that polls the Scheduler for algorithms that are ready to execute, launches them on the resources they are bound to, monitors their progress and reports their status back to the Scheduler. The current RVC dispatcher is a Tcl/Tk application.

Resources - computers registered with the Scheduler which can be used to execute plans. Currently the RVC can use any system running Unix as a resource.

Mass Store - file systems associated with resource machines where the RVC stores processed products.

All the parts of the RVC communicate with each other by passing commands and responses structured as S-expressions. S-expressions are easily generated and parsed by the different programming environments within the RVC (Lisp, C, C++, Tcl/Tk), and have enough structure to easily encode complex messages.

Example Product Generation A typical product generation within the RVC starts with a message like this, which is a request from an Access Client to generate a product of type `AVHRR1` starting from any products available from observation 350, and deliver the result to an anonymous FTP site on `my-host`:

```
(PROCESS
  (OID 350)
  (GOALS
    (GOAL (TYPE "AVHRR1") (FTP "my-host"))))
```

The Database takes the original message, adds in what it knows about observation 350, and forwards the augmented message to the PSD:

```
(PROCESS
  (ELEMENT (OID 350)
```

```
  (ATTRIBUTES
    (PLATFORM "NOAA-12") (SENSOR "AVHRR")
    (FILE "file://foo/mass/Q19971641043.L0_HRPT")
    (DATE (YEAR 1997) (MONTH 6) (DAY 13)
      (HOUR 12) (MINUTE 8) (SECOND 32))
    (ROWS 4497) (COLUMNS 2048)
    (BITS_PER_PIXEL 10)
    (STORED_BITS_PER_PIXEL 16))
  (EXISTING_TYPES
    (ALGORITHM (NAME "NOAA-12 AVHRR")
      (FILENAME
        "file://foo/mass/Q19971641043.L0_HRPT"))))
(PID 18)
(GOALS
  (GOAL (TYPE "AVHRR1") (FTP "my-host"))))
```

The PSD creates a plan to generate the requested product, allocates compute resources to execute it, and queues the resource-bound plan for execution, then informs the database that the product will be generated:

```
(MESSAGE (PID 18)
  (TYPE PROCESS) (STATUS SUCCESSFUL))
```

The PSD dispatches and monitors the processes of the plan as they run. When everything is finished, the PSD informs the database that the requested products are now available for anonymous FTP at the URLs in the FILES clause:

```
(MESSAGE (PID 18)
  (TYPE PROCESS_STATUS) (STATUS SUCCESSFUL)
  (FILES
    "ftp://my-host/pub/NOAA12_AVHRR1.gz"))
```

The database then e-mails the URLs to the requesting user.

How the PSD Works

Resource Registration The PSD views its domain as sets of *resources* and *algorithms*. Resources and algorithms are made available to the PSD by *registering* them: a curator clients sends the PSD an S-expression describing the new capability, the PSD unrolls it into an internal model, and then goes out across the network to gather anything else it needs to use the resource (probe storage sizes) or algorithm (copy and store executables).

A *resource* is a system the PSD may use to store files and run compatible programs. Resources are actually composed of a CPU type and one or more disk storage types. Here is an S-expression used to register a typical resource:

```
(NEW_RESOURCE (OID 113)
  (ETHERNET (HOST_NAME DANVILLE.GSFC.NASA.GOV)
    (IP_ADDRESS "202.170.170.8"))
  (FDDI (HOST_NAME DANVILLE-F.GSFC.NASA.GOV)
    (IP_ADDRESS "202.170.170.168"))
  (ARCHITECTURE
    (MAKE "HP") (MODEL "9000/735"))
  (OPERATING_SYSTEM
    (TYPE "HP-UX") (VERSION "B.10.10"))
  (ANONYMOUS_FTP (FTP_DIRECTORY "/pub"))
  (DATA_DIRECTORIES
    (DIRECTORY (NAME "/lost/rvc_online")
      (FILE_SYSTEM_TYPE ONLINE))
    (DIRECTORY (NAME "/mnt4/rvc_nearline")
      (FILE_SYSTEM_TYPE NEARLINE)))
  (EXECUTABLES_DIRECTORY "/usr/local/rvc/bin2"))
```

This description has one of everything the PSD considers to be a resource: a CPU (ARCHITECTURE and OPERATING_SYSTEM clauses), an area for anonymous FTP (used to make products available to the outside world), disk space for scratch storage while running programs (the ONLINE DATA_DIRECTORY) and disk space for long-term product storage (the NEARLINE DATA_DIRECTORY). The PSD takes this resource description and generates from it four scheduler resource descriptions (see below for examples).

```
(SCRIPT-ADDRESOURCE
  :PRETTY-NAME "danville.gsfc.nasa.gov"
  :NAME (COMPUTATION HP DANVILLE.GSFC.NASA.GOV)
  :ATTRIBUTES ()
  :AVAILABILITIES ((0 3616053101 1)))

(SCRIPT-ADDRESOURCE
  :PRETTY-NAME "danville.gsfc.nasa.gov:/scratch"
  :NAME (STORAGE DISK FILE-SPACE DSK22744)
  :ATTRIBUTES (DANVILLE.GSFC.NASA.GOV)
  :AVAILABILITIES ((0 3616053101 1997022)))
```

Each scheduler resource has a NAME in a hierarchical type space, a set of ATTRIBUTES in addition to its name, and a set of AVAILABILITIES describing how much of the resource can be used and at what times as a list of (*start-time end-time quantity*) specifications.

CPUs are named as (COMPUTATION *cpu-type domain-name*). Since the original resource description didn't specify an availability, the PSD assumes the resource is "always" available, from zero seconds to the scheduler's horizon, arbitrarily years from now. CPUs have one unit; as resources they are either free or busy. This model is simplistic, but not too bad given the nature of most image-processing code.

Disk storage is named as (STORAGE DISK *purpose unique-tag*). The domain names of any machine(s) that can access the storage are placed on the ATTRIBUTES list. Disk availability is measured in 1K byte blocks.

Resources are requested by specifying a (possibly incomplete) name, list of attributes, quantity, and duration; the scheduler then matches the specification against the available resources to come up with candidates. For example, a request for any HP CPU would have (COMPUTATION HP) as its NAME field.

Algorithm Registration and Planning An *algorithm* to the PSD is an executable program that takes typed input files and transforms them into typed out-

put files. Algorithms may have parameters and ancillary files. Here is an S-expression used to register a typical algorithm:

```
(NEW_ALGORITHM
  (ALGORITHM "navTIROS12")
  (ARCHITECTURE
    (MAKE "HP") (MODEL "9000/770"))
  (OPERATING_SYSTEM
    (TYPE "HP-UX") (VERSION "B.10.01"))
  (EXECUTABLE_FILENAME
    "exec://danville/mass/HP/RDCnavTIROS")
  (EXECUTABLE_SIZE 107108)
  (AUTHOR "Allen Lunsford (Hockey Legend)")
  (ARGUMENTS
    (ARGUMENT (USAGE INPUT_FILE)
      (DATA_TYPE "TIROSAUX12")
      (SWITCH "-iAUX")
      (PARAMETER "TIROSAUX12"))
    (ARGUMENT (USAGE INPUT_FILE)
      (DATA_TYPE "EPHEM")
      (SWITCH "-iEPHEM")
      (PARAMETER "EPHEM"))
    (ARGUMENT (USAGE OUTPUT_FILE)
      (DATA_TYPE "TIROSDROPOUT")
      (SWITCH "-oDROPOUT")
      (PURPOSE PRODUCT_GENERATION))
    (ARGUMENT (USAGE OUTPUT_FILE)
      (DATA_TYPE "GRID")
      (SWITCH "-oGRID")
      (PURPOSE PRODUCT_GENERATION))))
```

This description tells the PSD everything it needs to know to use this program:

- the name and location of the executable in the mass store if it needs to install it on a resource machine before running it (`EXECUTABLE_FILENAME` clause).
- the type of computer it runs on (`ARCHITECTURE` and `OPERATING_SYSTEM` clauses, identical to the ones in the resource description above).
- the number, order, and type of any arguments it takes (`ARGUMENT` clauses). The current PSD file typing system is deliberately simple: file types are symbolic tags in a flat type space. This turns out to be just enough information for a planner to be able to chain algorithms together.

The PSD takes this algorithm description and generates from it a SNLP plan step:

```
(DEFSTEP
  :PRECOND
    '((DATA-TYPE ?OID TIROSAUX12 ?I1120)
      (DATA-TYPE ?OID EPHEM ?I1130))
  :ACTION
    '("exec://danville/mass/HP/RDCnavTIROS"
      "-iAUX" ?I1120 "-iEPHEM" ?I1130
      "-oDROPOUT" ?O1140 "-oGRID" ?O1150)
  :ADD
    '((DATA-TYPE ?OID TIROSDROPOUT ?O1140)
      (DATA-TYPE ?OID GRID ?O1150)))
```

In the plan step, the `?I` variables represent input file names and the `?O` variables represent output file names; the `?OID` variable holds the object ID of an observation. The SNLP database fact `(DATA-TYPE` *observation-id file-type file-name*`)` means that *file-name* contains data of *file-type* derived from *observation-id*; thus the plan step says that given files of types `TIROSAUX12` and `EPHEM`, executing `RDCnavTIROS` will create files of types `TIROSDROPOUT` and `GRID`.

In addition to the plan steps generated from registered algorithms, the PSD has several hand-coded plan steps to handle moving typed files into and out of the mass store, and delivering them to anonymous FTP sites so users can retrieve them. Here, for example, is the plan step used to fetch a typed file from the mass store:

```
(DEFSTEP
  :PRECOND
    '((ON-SERVER ?OID ?TYPE ?CFILENAME))
  :ACTION
    '("gunzip-if-needed.script"
       -I ?CFILENAME -O ?FILENAME)
  :ADD
    '((DATA-TYPE ?OID ?TYPE ?FILENAME)))
```

And here is the step used to deliver a typed file to a FTP site:

```
(DEFSTEP
  :PRECOND
    '((DATA-TYPE ?OID ?TYPE ?CFILENAME))
  :ACTION
    '("ftptourl.script" ?CFILENAME ?FTPDEST)
  :ADD
    '((FTP-RESULT ?OID ?TYPE ?FTPDEST)))
```

`gunzip-if-needed.script` and `ftptourl.script` are shell scripts that can perform their function on any Unix machine; they need to be installed before a machine is registered with the PSD as a resource.

We now have enough plan steps to build complete plans to generate image products and return them to users. The PSD generates the planner initial conditions from a given observation *oid* by querying the database for the existing product types it has in the mass store for it; each of those types becomes a fact of the form:

`(ON-SERVER` *oid file-type filename*`)`.

A user request for a product of type *file-type* derived from *oid* turns into the goal clause:

`(FTP-RESULT` *oid file-type* "`ftp://`*host/path/filename*")

Handing the initial conditions and goals over to SNLP allows it to generate a DAG of plan steps.

Scheduling and Resource Allocation After SNLP generates its plan, the scheduler takes over and allocates resources to execute it. For each step in the plan, a scheduler task description is generated. Scheduler tasks can specify their minimum and maximum duration, their earliest and latest permissible start and

finish times, their relationship to other tasks, and the amount and types of resources needed while they execute. Here is a typical subtask description for the primitive PSD operation of fetching a file from the mass store:

```
(SCRIPT-ADDSUBTASK
  :OID 141486
  :TASK 14148
  :INVOCATION
    "exec://danville/gunzip-if-needed.script
      -i file://danvillev/incoming/Q19971641043
      -o /scratch/TASK-14148/TEMPFILE-17471"
  :PREDS (141481)
  :SUCCS (141487)
  :MIN-DUR 1
  :MAX-DUR 600
  :RESOURCE-REQUIREMENTS
    ((1823745
       (COMPUTATION)
         NIL 1 :UNIQUE)
     (1823746
       (STORAGE DISK FILE-SPACE)
         NIL 100000 :BEGINS)))
```

Note particularly the `RESOURCE-REQUIREMENTS`, which is a list of (*id-number resource-name resource-attributes quantity temporal-scope*) specifications. Since this task is a shell script, it can run on any computer (if it needed an HP machine, it would have asked for `(COMPUTATION HP)`), and it needs the computer only for the duration of the task (the `:UNIQUE` tag). It moves a 100 MB file from the mass store to local disk, and the file stays there for future tasks to use (the `:BEGINS` tag; the task that deletes this file will have a specification with a `:ENDS` tag to free the resource).

Once all the plan steps have scheduler tasks associated with them, the scheduler is turned loose, and it tries to match the resource requirements with the resources it has registered. If it succeeds, the PSD can walk the resulting resource specifications and use them to fill in the blanks in the plan, mainly generating temporary directory and file names for intermediate results. At this point, any tasks with no predecessors are marked in the temporal database as ready-to-run.

Dispatching and Execution The PSD dispatcher is a Tcl/Tk application that polls the scheduler at fixed intervals (ten seconds or so), telling it to update its clock and asking it if anything new is ready-to-run. Tasks that are ready are launched on the resources bound to them (via remote shell call in most cases) with a small Tcl/Tk process monitoring them as they run. This execution monitor process sends messages to the scheduler to tell it when tasks start, when they finish, and whether they completed successfully or not (judged solely on their exit status code; registered algorithms must report their status correctly or risk confusing the scheduler).

When a task completes successfully, its successor tasks are marked as ready-to-run, and will be launched when the dispatcher next polls for them. When all the tasks of a plan complete successfully, the scheduler notices and sends a `PROCESS COMPLETE` status message. If a task executes for longer than its `MAX-DUR`, the scheduler will notice when the dispatcher updates its clock; it can then report the error.

Experience with Current System

The current fielded RVC system was created in rapid prototyping mode by integrating several prototype systems that have been under development in Code 935 since 1989. Our goal was to get a working system into the hands of real users as quickly as possible, to get feedback from them to guide future development. As could be expected with any prototype, we had to make design compromises and work around limitations in our tools to get the system out the door. We discuss some of those problems and our solutions to them below.

Planning SNLP turned out to be a useable tool for the simple plans we had to generate for RVC product generation. SNLP itself accounts for about half of the PSD's cycles. Much of this is due to the fact that SNLP is implemented for students to study and experiment with; it is coded for clarity more than high efficiency.

In particular, SNLP will sometimes run off and do much more work than necessary to generate plans, and there is no easy way to make it more efficient. For example, given a goal set asking for N file types and a plan step that had those types in its outputs, SNLP will generate N candidate plans, one for each output file. These plans look equally good to SNLP, so it will keep them all, and generate $N \times (N-1)$ plans at the next stage, ultimately ending up with $N!$ identical plans.

Scheduling The DADS-based scheduler performs quite impressively. The Honeywell TMM core algorithms are very efficient; scheduler resource allocation takes up at most one fourth of the PSD's cycles - about five seconds overhead on a typical twenty minute image product job.

The biggest problem we had with the scheduler was that we used it in a multi-threaded Lisp environment, and it wasn't designed with that in mind. The problem was that asynchronous events (polling from the dispatcher, status reports from the execution monitor) needed to access and update the resource temporal database, and there was no locking on that data structure. Often what looked from the outside like simple requests, like asking a task what its earliest start time was, turned out to require exclusive access to the temporal database. The PSD was failing intermittently for months until we realized this, and only became stable after we set up a process-lock and made scheduler-using tasks acquire it before doing anything dangerous.

Costs and Benefits As we started out, there was some worry that full-blown AI-based planning and

scheduling would be too expensive computationally - that the PSD would consume a significant fraction of a resource machine's memory and cycles for little benefit. This has turned out not to be the case.

The cost of having planning and scheduling in the RVC is small relative to the cost of running a typical image processing job. The Allegro Common Lisp executable containing the PSD is the largest single program in the RVC system, but it has turned out to be a well-behaved application and a good neighbor as far as system paging is concerned, even without any serious tuning. The system takes about twenty seconds to plan, schedule, dispatch and monitor a typical process request generating a basic set of image products from a TIROS or GOES data file; running the entire process takes about twenty minutes, so the overhead of the PSD is not significant.

The benefits of having planning and scheduling in the RVC are currently not as large as they could be, but are expected to grow as the system matures and is used to solve larger problems. The planner and scheduler are both underutilized in the current system.

The planner currently solves problems that are mostly within the capabilities of simpler tools like the Unix `make` utility. The main things the RVC gets from using a planner now are expressiveness and tolerance of ambiguity: it is safe to register algorithms that allow more than one way to create a given type of file, because the planner can be relied on to choose the one that is best according to its search strategy. The current system relies on this internally. When a product derived from a given observation is requested, the system can deliver it to an FTP site for pickup in several different ways:

- If the product has never been requested before, it can run algorithms to generate it from precursors in the mass store, then copy it to an FTP site.
- If the product is already in the mass store, it can copy it directly from there to an FTP site.
- If the product is already at an FTP site, it can just tell the user where it is.

The planner chooses an appropriate delivery method by selecting the plan with the fewest steps that works given the initial conditions for the observation.

The scheduler currently uses very coarse estimates for algorithm time and space utilization; this means the scheduler's estimates for how long a plan will take to run or how much disk space it will need to finish are educated guesses at best.

Future Plans

The rapid prototyping approach taken to develop the RVC system appears to have succeeded. The first release of the prototype RVC system was designed and implemented in about nine months. A full-fledged implementation, incorporating feedback from the original RVC sites as well as other improvements, should be completed and deployed at the various RVCs by the middle of 1998.

A number of enhancements are slated for the planner/scheduler module. Both the planner and scheduler could do more in the RVC given more information about their domains. The planner could be used to generate plans from fuzzier specifications of output requirements, if it had a better model of what algorithms do. The scheduler could generate more accurate estimates of resource utilization if it had a better model of how algorithms use resources.

One of the most significant (and most difficult) improvements will be in the addition of a hierarchical typing system throughout the IIFS, replacing and extending the current simple symbolic typing scheme. This will require each output type to produce an accompanying metadata file which will be maintained by the database. The planner will be able to pass specific attributes of this metadata to algorithms further along in the chain when the product is part of an intermediate step in producing a goal.

We are also cleaning up the PSD internally, adding abstract interfaces to planning, scheduling, and dispatching, to make it easier in the future to change planning or scheduling tools if needed. We already intend to try a different planner in the next major release, Lansky's action-based planner COLLAGE (Lansky 1994).

We also need to improve the execution monitor, so it can gather better resource utilization data. At present, the scheduler is under-utilized in the PSD, primarily because we currently can't give it good data about algorithm resource use. Having this data will be critical for one of our long-range goals, which is to allow RVC sites to share resources and cooperate in processing data.

The biggest hurdle in making these enhancements will be keeping them intelligible to the expected RVC user community. We expect to have around sixteen RVC systems out in the field by the time of the next major release; the users will be experts in remote sensing, and must not be required to become experts in computer science or AI to use their RVC.

References

Cromp, R. F.; Campbell, W. J.; and Short, Jr., N. M. 1993. An intelligent information fusion system for handling the archiving and querying of terabyte–sized spatial databases. In *International Space Year Conference on Earth and Space Science Information Systems*, 586–597. American Institute of Physics.

Lansky, A. 1994. Action-based planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems.* American Association for Artificial Intelligence.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of the Ninth Na-*

tional Conference on Artificial Intelligence, 634–639. American Association for Artificial Intelligence.

Short, N. M.; Cromp, R. F.; Campbell, W. J.; Tilton, J. C.; LeMoigne, J. L.; Fekete, G.; and Netanyahu, N. S. 1995. Mission to Planet Earth: AI views the world. *IEEE Expert* 24–34.

White, J.; Boddy, M.; Nelson, K.; Atkins, R.; and Bedet, J. 1996. Programmers guide to SWAMPANGEL scheduling. Technical Report software version v3-02, Honeywell Technology Center.