SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2007)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.845

The benefits of posing application software as a language interpreter

W. Van Snyder*, †

Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 183-701, Pasadena, CA 91109, U.S.A.

SUMMARY

Complicated and comprehensive software that is meant to execute in a non-interactive or semi-interactive mode needs to be configured to carry out the desired tasks, needs to carry out those tasks efficiently, needs to be extensible to take on additional ambitions, and needs to be maintainable. To reduce costs, it is helpful if experts in the discipline to which the program applies can configure and operate the program without needing to become expert software engineers and without needing to become familiar with the internal details of the program, and if software engineers who develop and maintain the program need not become experts in its target discipline. Progress toward these goals can be advanced by posing the software as a language interpreter. We describe the application of this principle to ground-based data analysis software for the Microwave Limb Sounder instrument on the NASA Earth Observing System Aura satellite, but we believe the principle has substantially broader applicability.

Received 9 November 2006; Revised 8 June 2007; Accepted 26 June 2007

KEY WORDS: mathematical software; Fortran; scientific software; real-time systems; operating systems

INTRODUCTION

Ground-based data analysis software for the Microwave Limb Sounder (MLS) instrument on the Earth Observing System (EOS) Aura satellite [1,2] (as opposed to software that executes within the instrument or spacecraft) is divided into four major programs. The second of these, called *Level 2*, or more tersely *L2*, is charged with the task of analyzing observed spectra of microwave

*Correspondence to: W. Van Snyder, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 183-701, Pasadena, CA 91109, U.S.A.
†E-mail: van.snyder@jpl.nasa.gov

Contract/grant sponsor: National Aeronautics and Space Administration

thermal emission (*radiance*) from the atmosphere, altogether 660 million observations, to deduce its temperature and the concentrations of approximately 20 trace constituents at roughly 250 000 points, that is, roughly 5 million results, every day.

As one might expect, extensive computations are required, but numerous factors contribute to the organization and progress of the computation. In addition to the radiance, a spectroscopy catalog, antenna patterns, filter shapes, and orbit and attitude data, an initial guess for the parameters of interest, and a few other minor data such as leap seconds are required. Radiances in some spectral bands are useful to observe some molecules but irrelevant to others, or are useful at some altitudes but not at others. Emission from different molecules can most efficiently be modeled by different computational methods. Finally, the same primary model that is inverted to determine the temperature and composition of the atmosphere can be used 'offline' for scientific investigations unrelated to instrument data analysis, and to calculate the derivatives of radiances with respect to temperature and concentration, for use by other models that can be deployed to deduce composition and temperature from radiance.

Other than the radiance, orbit and attitude data, and initial guess (which is obtained from weather, climate, and chemical transport models, and climatological averages), all of the considerations arising from the foregoing could be incorporated into the L2 program as initial data and ordinary program decision making. From our experience with a previous instrument and its data analysis software, we learned that configuring the software to operate efficiently, reliably, and accurately requires extensive experimental tuning. Statements in a general-purpose programming language are a very low-level representation of the configuration of the program. It would further be difficult to represent and maintain the configuration as a coherent document, disentangled from all the minutiae of the program. As such, if the configuration were expressed by ordinary initial data and decision-making statements in the program, it would be tedious to change and more error prone than if it were expressed at a higher level. Perhaps more importantly, since the size of the L2 program now exceeds 250 000 lines, recertification after each tiny tweak of the configuration could be hideously expensive.

Therefore, we chose to configure each of the major MLS programs from one of their inputs rather than using program statements. In a bygone era, that input might have been a sequence of numbers, carefully organized in a rigid format and sequence, thereby being difficult to develop and maintain. Tuning a configuration expressed in that way would have been nearly as difficult as tuning one expressed within the program. An inch further along, one might have used something less rigid, such as Fortran NAMELIST. Neither of these methods addresses, in a convenient way, a need to create indefinite numbers of objects or to specify interrelations among them. Fortunately, today we have more computational resources at our disposal and more software technology upon which to draw, which allows us to pose the configuration specifications as easily readable, writable, and maintainable specifications, oriented toward the problems to be solved rather than the details of how to solve them.

The notation for the configuration follows a grammar, and it is analyzed using conventional compiler technology. Therefore, it can be thought of as a 'little language' or a 'domain-specific language' in the spirit of YACC [3]. The notation is not a general-purpose programming language, and a configuration expressed in that notation is not a program, just as a grammar expressed using YACC is not a program. The notation declares details of the configuration, such as which radiances to use and which results to obtain, and specifies actions to be performed, such as to solve for specified quantities using specified radiances and methods. The program is organized as an interpreter of

DOI: 10.1002/spe

this language. When it encounters a declaration specification, it builds a data structure. When it encounters an action specification, it performs the specified action. The fact that our problem does not require the notation for its configuration to provide for iteration or testing is not relevant to the concept. Other problems might require these capabilities. Syntax to denote them and methods to process them are easily incorporated, a fact that illustrates the power of the paradigm.

A typical operational configuration is about 13 000 lines, of which roughly 10 000 are generated automatically from roughly 500 statements by a macro expansion package. Another roughly 500 statements describe the instrument, which has not (cannot!) been changed since the spacecraft was launched[‡]. This leaves roughly 3000 lines that configure and specify the computations proper. Since the program and its configuration are orthogonal instead of intertwined, certifying one is largely independent of certifying the other. The size of the configuration is only about 1.2% the size of the program, so certifying it is of much lower cost than certifying the program. By analogy, ask yourself, 'Do I certify my compiler every time I change my program, or do I just certify my program?'

SUPERFICIAL DESCRIPTION OF COMPILER TECHNOLOGY

We use conventional techniques borrowed from compilers to analyze the configuration. The characters of input are grouped into *tokens*, analogous to words and punctuation marks in natural languages, by a process called *lexing*. The structure of sequences of tokens is then recognized by a process called *parsing*, which produces an *abstract syntax tree* (Figure 1), analogous to a sentence diagram resulting from grammatical analysis of a sentence in natural language. Just as with natural language, it is possible in synthetic languages to express perfectly grammatical nonsense, so the first step after constructing the tree is a sanity check.

The interested reader is referred to a text specifically concerned with compiler technology, e.g. [4,5], for more details than those presented here.

Syntax

The configurations for all four of our major programs share a very simple syntax: It consists of begin–end blocks, each containing specification statements. The block organization is used to indicate which parts of the configuration are the same for every batch of data (such as the receiver description or the spectroscopy database), and which parts are re-traversed for each batch of data (substituting for explicit loop control statements). Each specification optionally begins with a label followed by a colon, and thereafter consists of a word followed by a list of zero or more name–value pairs, each pair preceded by a comma. The blocks and specifications that are particular to each program are different. A short example of some specifications appears in Appendix A. Others might prefer a different syntax, while preserving the essential idea of separating the program from its configuration, and specifying the configuration by using a domain-specific language.

[‡]Specifying the instrument as part of the configuration rather than within the program allows the program to be used to investigate other instruments, including proposed instruments that have not yet been constructed or deployed.

DOI: 10.1002/spe

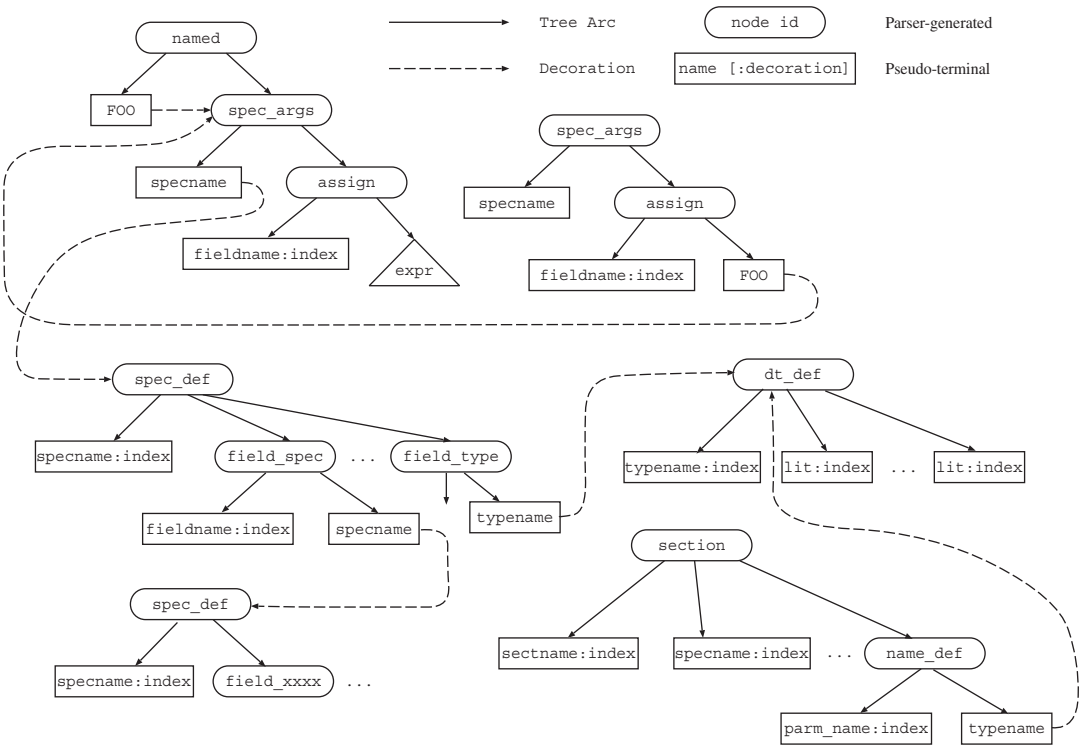

Figure 1. Some subtrees of the abstract syntax tree with decorations.

Lexing and parsing

Lexing is carried out by a handwritten deterministic finite automaton. Each token consists of its *part of speech* (name, number, plus sign, etc.), the index of its text in a *string table* (so no further searching is needed), and the position where it appeared in the configuration (for error reporting).

One could use a parser generated automatically by a parser generator, e.g. YACC [3], or a handwritten recursive-descent parser. Our grammar is quite simple, so we chose the latter to avoid the need for dependence upon yet another program.

To facilitate type checking, and since parts of the configuration need to be examined more than once, the result of parsing is represented by a tree. In the case of MLS L2, we traverse part of the tree several times, once for each batch of data (about 2400 spectra, from which the temperature and composition are deduced at about 1400 points in the atmosphere).

Type checking and declaration analysis

Syntax rules allow the value of each field to be a number, a word, a more general expression, or a string, but if a field's value is expected to be a numeric expression, then a string or the label

DOI: 10.1002/spe

of another specification is not appropriate (i.e. it is grammatically correct nonsense). A benefit of posing software as a language interpreter, which we see as a particularly important benefit, is that a single framework can be used to specify and check the names of blocks and their allowed order, the names of specifications that can appear in each block, the names of fields in each specification, which fields are required, whether fields can be duplicated, the type of each value for each field in each specification, the relationship between configuration items, and to specify, check, and convert the physical units of numeric inputs to standard units (e.g. kilometers or megaHertz, even if the inputs are meters or gigaHertz).

The details of how the L2 program represents type-checking requirements and how it does type checking are less important than the principle that type checking ought to be automatic and unavoidable, and performed in one place in the program. A superficial description of our method appears in Appendix B. The interested reader is referred to a compiler text for more information.

Checking of physical units of numeric values could have been incorporated into the type-checking framework, but the importance of doing so was not appreciated at the time of its design; since other development has been of higher priority, it has not yet been so incorporated, so each process that evaluates a numeric expression is responsible for units checking (conversion to standard units is done automatically within the expression evaluator). Units analysis is a by-product of expression evaluation, and units checking is therefore quite simple. Even so, some of our developers are less assiduous than others regarding units checking. In hindsight, we recognize (and recommend) that units checking should have been incorporated into the type-checking process.

TRAVERSING THE TYPE-CHECKED TREE

After type checking is complete, the actions proper of the program are carried out by traversing the tree again, skipping the type definitions, but repeatedly traversing a large part of the tree, once for each batch of data. The program uses the index of the name of each block or specification as a selector to carry out an action. The index for each name is assigned during the type-checking phase.

The coarsest-scale structure of our configuration is a `begin ... end` block, so subtree roots near the root of the tree represent the block structure of the configuration. The first procedure in the tree traversal invokes the next levels of processing depending on the block indices and includes a looping structure to process some of the blocks repeatedly. This is what allows to escape the requirement for explicit loops and tests within the configuration.

Assuming `root` is the index of the part of the tree that arose from input, the following illustrates the processing of sections. The `decoration` function accesses a field of the same name in the specified tree vertex, which in the case of a section subtree is the section index, thanks to the type checker. Decorations are discussed in Appendix B.

```
do i = 1, nsons(root)
 son = subtree(i, root)
 select case ( decoration(subtree(1,son)) )
 case ( z_globalSettings )
  call set_global_settings ( son, ... )
  ...
```


DOI: 10.1002/spe

```
  case ( z_spectroscopy )
   call spectroscopy ( son )
   ...
  case ( z_construct, z_fill, z_join, z_retrieve )
   do ! Loop over the chunks of data
    select case ( decoration(subtree(1,son)) )
    ...
   end do
  end select
 end do
```

To add a section, one need only add a branch to the case selector control structure and write the handler for that section, which (if one has been careful about software engineering practices) is largely independent of the others.

The procedure that processes each section knows what is allowed in that section and knows that only allowed specifications appear, thanks to the type checker. For example, the `fill` section is allowed to have a `vector` specification, among others, and is not allowed to have a `retrieve` specification. Each procedure that processes a specification uses the fields in the specification to complete its definition. It knows what fields are allowed, knows that required fields have been verified to be present, knows that the types of field values are correct, and knows that disallowed fields do not appear, again thanks to the type checker.

Assuming that `root` is now the index of the root of a section subtree, the following illustrates the processing of that section and one specification within it. The `get_spec_id` function knows how to skip the optional label and then accesses the decoration of the specification subtree, which is in this case the specification index, again thanks to the type checker. The `get_field_id` function works similarly. The `decorate` subroutine stores a value into the specified tree vertex.

```
 do i = 2, nsons(root)-1 ! Skip the section name at begin and end
  son = subtree(i,root)
  select case ( get_spec_id(son) )
  case ( s_vector )
   do j = 2, nsons(son) ! skip spec name, get to fields
    gson = subtree(j,son)
    select case ( get_field_id(gson) )
    case ( f_template )
     templateIndex = decoration(decoration(subtree(2,gson)))
    case ( ... )
    end select
   end do
   call decorate ( son, addVectorToDatabase(vectors, createVector( &
    & vectorName, vectorTemplates(templateIndex), ...) ) )
  case ( ... )
   ...
  end select
 end do
```


DOI: 10.1002/spe

This illustrates that processing a `vector` specification creates a vector, adds it to a database, and decorates a tree vertex within the specification with the vector's index in the vector database. When a field for which the value is required to be a vector is processed, the decoration of the vertex in the tree where the vector is referenced is known to be the index in the tree of the vector's declaration (thanks to the type checker), whose decoration in turn is the index of the vector in the vector database. The `template` field of a `vector` specification is required, and is required to be the label of a `vectorTemplate` specification (analogous to a Pascal type definition), which specifies the quantities the vector contains. Thanks to the type checker, this procedure knows that the `template` field's subtree is decorated with the index of the template. A `vectorTemplate` specification is processed similarly to a `vector` specification. The root of its subtree is decorated with the index of the created template. The procedure to process another specification would determine the index of a vector in the vector database in the same way as the `templateIndex` above is accessed.

It is clear that the program can be configured to deal with any desired number of vectors, each with any desired collection of quantities (which are in turn defined by declarations within the configuration specification); this flexibility extends to all of the data structures to which the configuration specification has access.

Some of the specifications, such as `vector`, simply result in construction of a data structure; others specify an extensive computation. For example, a `retrieve` specification results in invoking a Newton method to invert the forward model. A retrieval can act on several forward model configurations, which are specified by `forwardModel` specifications, whose labels are mentioned in the `retrieve` specification. An example of a `forwardModel` specification, a `retrieve` specification, and specifications for some related data structures is shown in Appendix A.

Adding specifications to a section, or fields to a specification, requires simple additions to the type-checking tables (illustrated in Appendix B), adding branches to the case selectors, and either putting simple processing in line or writing the appropriate procedures, which again (assuming careful software engineering) are largely independent.

The MLS L2 program typically spends 94% of its time evaluating the forward model and its derivatives with respect to composition and temperature, and nearly all of the remaining 6% doing linear algebra related to inverting the model, both of these processes, operating together, being triggered by `retrieve` specifications. L2 is an example of those programs that perform intensive computations that are richly parameterized, in which many different variations of the shared parameterized computations occur in the same application, and in which each invocation of such a computation consumes substantial computational resources. These computations rely on parameters and data structures specified by the configuration, and are triggered by statements in the configuration, but are not themselves organized as interpreters.

Processing specifications that declare parameters, data structures, and their relationship with one another is not very computationally expensive. When specifications that cause substantial computational processes to be invoked are encountered, the efficiency of the code that traverses the tree and invokes them is irrelevant: the overhead of interpretation, and indeed of processing every specification other than `retrieve`, is easily offset. The MLS L2 program typically spends 15 s reading and parsing the configuration, creating, type checking, and traversing the tree, and executing all actions other than `retrieve` specified by the configuration. A typical run requires between 15 and 30 h, so the overhead of interpreting a domain-specific language varies between 0.015 and 0.03%: The interpretive approach comes essentially for free.

DOI: 10.1002/spe

BENEFITS FOR PROGRAM DEVELOPMENT AND MAINTENANCE

In addition to automatic and complete type checking, several other benefits accrue as a consequence of posing the program as a language interpreter. First, of course, is that the type checking is concentrated in one process, so if errors are discovered they can be corrected just in that place. If a program carries out similar processes in several places, they might be slightly different, and subtly differently incorrect, in each place. Thus, concentrating type checking in one place reduces the chance for difficult-to-find subtle errors, thereby increasing reliability and reducing maintenance cost. Furthermore, this does not require every developer to understand how to do type checking. Finally, the syntax analysis and type-checking framework are reusable since the syntax is the same in each of our programs and the type-checking requirements for each program are specified separately from the framework itself, using an organization that is common to our several programs. Indeed, this component is sufficiently abstract that it could be far more broadly applied than only to our four programs.

As new ambitions for the programs arise, it is frequently possible to respond to them by adding or extending specifications for facilities to which the configuration already has access. In the examples in the previous section, this would entail new branches in the `select case` control structures. When it is necessary to extend or expand the facilities that can be accessed by the configuration, it has proven to be easy to do so either by allowing new values in existing fields (usually new literals for enumeration types), new fields for existing specifications (and sometimes new types for their values), or new specifications. In most cases, the new facilities have been added by minor modifications of existing procedures or by entirely new procedures that are nearly independent of other procedures.

By way of example, the MLS L2 program was originally envisioned to use only a 'full' forward model (using line-by-line evaluation of the spectroscopy) and to invert that model to produce composition and temperature. We soon realized we could use simplified models for certain chemical species. These simplified models use results computed offline by the full model. Our ambitions also increased to encompass cloud modeling. Each of these modifications, including using the full forward model offline to produce coefficients for later use online by simpler models, was easily incorporated, because posing the program as a language interpreter has uncoupled parts one from another, allowing modifications to be implemented at relatively lower cost than would be the case with a more coupled design or a structure less uniform than that required by the interpreter paradigm.

BENEFITS FOR STAFF ALLOCATION AND TRAINING

As experience with the MLS L2 program is gained, it is becoming clear, as anticipated, that performance, in terms of both running time and accuracy of results, can be improved by tuning its configuration. Tuning the configuration depends upon knowledge of atmospheric chemistry, radiative transfer, and characteristics of the instrument, and is largely carried out by specialists in those disciplines, not by software engineers. If the configuration were represented within the program, it would have been necessary for those specialists to become familiar with the structure and methodology of the program, and the programming languages in which it is expressed, and to know where throughout the program the small fraction they were expected to tune could be found

DOI: 10.1002/spe

(and to keep their hands off the rest!). Even though the specialists who configure the program work with a configuration that specifies only what is to be done rather than every detail of how every step of every computation is done, this tuning is a time-consuming process. It would be far more difficult and far more time consuming if the configuration specification were more rigid, or worse, entangled within and scattered throughout the program.

Posing the program as a language interpreter provides flexibility, readability, modifiability, and maintainability of the configuration specification necessary for specialists to configure the program; yet, it does not require them to become expert software engineers or to become familiar with the internal operational details of the program. Similarly, the project software engineers, even those responsible for the forward model, have not found it necessary to become experts in the disciplines of atmospheric chemistry and radiative transfer. We believe this observation is applicable in many disciplines, not only in data analysis software for satellite-borne instruments or in scientific and engineering software.

CONCLUSIONS

Posing a program as a language interpreter confers several benefits. First, it allows to check easily whether values in the configuration specification have the correct types and units, and the correct relationship with one another. Second, it concentrates type checking in one place, thereby reducing development and maintenance cost, and increasing reliability. Third, many of the facets of the specification can be processed more independently from one another than might be the case with a more rigid structure. Fourth, it allows modifications to the program to be implemented more independently than might be the case with a different organization, thereby reducing development and maintenance costs. Fifth, providing a problem-oriented abstraction separates configuration of the program from the program proper, which allows members of the team to concentrate on their fields of expertise, and thereby substantially reduces development, certification, configuration, and training costs. Sixth, it confers considerable flexibility on the organization of the configuration, and thereby on the operation of the program. Seventh, it simplifies the program by using one data structure to represent and check the configuration, and to specify what is to be done and when to do it. Eighth, it adds negligible overhead cost compared with the primary mission of the program. Finally, it deploys well-known existing compiler technology that has a well-understood theoretical and mathematical foundation, in an easily reused framework.

APPENDIX A. SHORT EXAMPLE

This example is a tiny fraction of a real configuration. It illustrates the appearance of a configuration and how the specifications can be related to one another. '$' at the end of a line indicates continuation. `/xyz` is the same as `xyz=true`. Brackets enclose arrays. A colon within a field value (except within strings) was intended for ranges, but can be used for arbitrarily related pairs of numeric expressions. Semicolon precedes a comment. The fields shown for each specification are usually only a subset of the available fields. The program assumes documented default values for omitted optional fields.

DOI: 10.1002/spe

```
vGridStandard: vGrid, type=Logarithmic, coordinate=Zeta, $
   start=1000mb, formula=[25:6,12:3] ; 25 at 6 per decade, 12 at 3 per decade
hGridStandard: hGrid, type=regular, spacing=1.5 degrees, origin=0 degrees, $
   module=GHz
CH3Cl: Quantity, vGrid=vGridStandard, hGrid=hGridStandard, type=vmr, molecule=CH3Cl
stateTemplate: VectorTemplate, quantities = [ temperature, BrO, CH3Cl, CH3CN, ... ]
state: Vector, template=stateTemplate
fullextFwmR3: ForwardModel, type=full, phiWindow=5 profiles, $
   moleculeDerivatives=[extinction], /atmos_der, $
   integrationGrid=vGridTangent, tolerance=0.2 K, $
   /do_conv, /allLinesforRadiometer, signals=['R3:240.B33W:O3.C3'], $
   molecules=[extinction, [ O2, O2, O_18_O ], $
   [ O3, O3, O3_R3, O3_V1_3, O3_V2, O3_ASYM_O_18, O3_SYM_O_18 ]]
 Retrieve, state=xR3extinction, fwdModelExtra=state, $
   measurements=yR3extinction, measurementSD=yNoiseR3extinction, $
   apriori=aR3extinction, covariance=SaR3extinction, $
   outputSD=sdOutR3extinction, fwdModelOut=fR3extinction, $
   aprioriFraction=aprioriFractionR3extinction, $
   lowBound=lbR3extinction, highBound=hbR3extinction, $
   diagnostics=diagR3extinction, forwardModel=fullExtFwmR3, $
   columnScale=norm, maxJ=10, lambda=0.0, Ftolerance=0.01
```

The `vGrid` specification defines a 'vertical grid' labeled `vGridStandard` of logarithmic type with coordinate ζ, which is $-\log_{10} P$, where P is pressure in millibars. The grid starts at 1000 mb, and then has 25 levels at six per decade of pressure, followed by 12 levels at three per decade.

The values of the `type` and `coordinate` fields are verified by the type checker to be literals of specified enumeration types. The values of the `start` and `formula` fields are verified by the type checker to be numeric. The subprogram that processes the specification verifies the units of the `start` field and that the value of the `formula` field is an array of tuples of unitless numbers. Similar considerations apply to the other specifications illustrated here.

The `hGrid` specification defines a 'horizontal grid' labeled `hGridStandard` that has regular spacing of 1.5° (this is orbital angle, not longitude or latitude) starting at 0°. It is related to the gigaHertz module of the instrument.

The `Quantity` specification defines a portion of the state vector related to the molecule CH_3Cl. This is a volume mixing ratio quantity represented on the two previously specified grids. The values of the `vGrid` and `hGrid` fields are verified by the type checker to be the labels of `vGrid` and `hGrid` specifications, respectively.

The `VectorTemplate` specification defines a template for vectors (similar to a Pascal-type definition) labeled `stateTemplate`. It contains quantities which the type checker verifies are defined by `Quantity` specifications, for temperature, BrO, CH_3Cl, CH_3CN, etc.

The `Vector` specification defines an instance of a collection of quantities, as specified by the referenced template, labeled `state`.

The `ForwardModel` specification defines a configuration of the forward model labeled `fullextFwmR3`. In this case, it specifies using the full forward model to examine five profiles of data, to calculate derivatives of radiance with respect to extinction, to compute mixing ratio derivatives, to carry out the integration on a specified grid named `vGridTangent` (not shown), that the tolerance for radiance calculations within the forward model is 0.2 K (radiance is measured

DOI: 10.1002/spe

in brightness temperature), that antenna convolution is to be performed, that all spectral lines from the spectroscopy catalog whose center frequencies are within the frequency bands measured by the radiometers specified by the `signals` field are to be considered, and that calculations are to be carried out for the specified molecules.

The `Retrieve` specification causes retrieval of atmospheric quantities mentioned in the vector labeled `state` using measurements from the vector-labeled `yR3extinction` with standard deviations in the vector `yNoiseR3extinction`, and *a priori* values for the retrieved quantities in the vector `aR3extinction` with covariance `SaR3extinction`. The solution is to be placed in the vector `fR3extinction` and its calculated standard deviation is to be placed in the vector `sdOutR3extinction`. The fraction of the solution that is due to *a priori* values is to be reported in the vector `aprioriFractionR3extinction`. The low and high bounds for the solution, which keep the retrieval within physically meaningful limits, are given by vectors `lbR3extinction` and `hbR3extinction`, respectively. Diagnostic quantities that are of interest to study how the retriever converges to a solution are to be reported in a vector `diagR3extinction`. The forward model configuration to be used for this retrieval is given by a `ForwardModel` specification labeled `fullExtFwmR3` (the one shown above). The Jacobian matrix used during the Newton method is to have its columns scaled to unit length (which minimizes the κ_2 condition number). The Newton method is to start with a Levenberg–Marquardt stabilization parameter of zero, and is to continue until either 10 iterations have been taken, or the norm of the difference between computed and measured radiances is 0.01 K.

The details of how the several vectors are filled or the details of the disposition of the computational results are not shown. Some of the results are output products; some are used as input to a subsequent stage of computation.

APPENDIX B. A FEW DETAILS ABOUT TYPE CHECKING

A brief discussion of the methods used by the MLS programs for type representation and checking follows. The details are intended to be illustrative rather than essential. Other programs that follow the basic principle of concentrating type checking in one place might use an entirely different system.

The MLS programs encode enumeration types and their enumerators, and the type-checking requirements, using a tree. The type-checking tree is part of the same data structure as the abstract syntax tree. It is built by program statements before the parser runs, not within the configuration, as it does not change frequently, but it is quite simple to change, and is not scattered throughout the program. After the parser runs, the type-checking tree and the abstract syntax tree are joined into a single tree by a new root node.

The type-checking process then traverses the entire tree, aided by a *declaration table*, into which declarations of words that consist of references to parts of the tree are entered. Leaves are connected to the declaration table by their string index. In addition to connections that represent the sentence structure, vertices in the tree may be connected to each another using a field in each vertex called the *decoration*, which is also used for other purposes since it is just an integer. When the value of a field in a specification is the label of another specification, the tree vertex of the reference is decorated with the index of a tree vertex within the declaration subtree for that label.

DOI: 10.1002/spe

The type-checking process uses the structure of the type-specification tree to verify that the input meets the type-checking conditions.

This unified type-checking system relieves the remainder of the program of this responsibility. If type checking were not carried out by a single unavoidable process, for which one developer is responsible, each module of the program would have to do this on its own. This would be repetitive, and would further require training all of our developers in this technique, and enforcing a discipline of incorporating it. The quality and effectiveness of this would vary greatly from one module to another.

The subtrees shown in Figure 1 illustrate the relationship between subtrees arising from inputs and the type-checking subtrees for those inputs. The subtrees with roots `spec_def`, `dt_def` and `section` are part of the type-checking requirements, which would be 'to the left' of the remaining subtrees that would arise from input. The decorations (shown as dashed lines) are inserted during the type-checking traversal of the tree. Since the part of the tree that specifies type-checking requirements is 'to the left' of the input and the tree is traversed depth first, left-to-right, the interrelations encoded by the decorations within that part are inserted by the time the part of the tree that arises from input is traversed.

The input giving rise to the top left `spec_args` subtree would be of the form `foo:` *specname*, *field*=*expr*, which defines a specification labeled `foo`. The input giving rise to the `spec_args` subtree at the top right would be of the form *specname, field*=`foo`, which references a specification labeled `foo`. Notice that the `foo` vertex in the top right subtree is decorated with the index of the `spec_args` vertex in the top left subtree. This vertex was found because its index had been entered into a declaration for `foo` when the top left subtree was processed.

The `spec_def` subtree at the middle left shows the type-checking requirements for the top left subtree. Notice that the `specname` vertex of the top left subtree is decorated with the index of the `spec_def` vertex of the middle left subtree. This vertex was found because its index was entered into a declaration for `specname` when the type-checking subtrees were processed. The `field_spec` subtree of the `spec_def` vertex indicates that the value of the field named by its first son is required to be the label of a specification whose `specname` is labeled by its second son. The decoration of the second son indicates this by being the index of the bottom left `spec_def` subtree. A `field_type` subtree can specify numeric, string, boolean, or enumeration types. The `field_type` subtree of the `spec_def` vertex indicates that the value of the field named by its first son is required to be of the enumeration type specified by the `dt_def` subtree at the middle right. A particular literal can be a literal of several enumeration types. The `section` subtree at the bottom right indicates that the section can contain a parameter definition for which the value is required to be of the type specified by the `dt_def` type definition at the middle right, as indicated by its decoration.

In addition to being the index of another subtree, decorations can have other uses. For example, the decoration of a `spec_def` vertex can indicate that none of its fields can be duplicated, while the decoration for a `field_spec` or `field_type` vertex can indicate that the field is required.

As mentioned above, the type-checking subtrees are built by program statements before the parser runs. Here is an example of a definition of the enumeration type `fwmType`:

```
begin, t+t_fwmType, l+l_baseline, l+l_linear, l+l_full, l+l_scan, &
    l+l_scan2d, l+l_cloudFull, l+l_hybrid, l+l_switchingMirror, &
    l+l_polarLinear, n+n_dt_def, &
```


DOI: 10.1002/spe

This is just a list of integer expressions. *begin* means 'begin a subtree;' $t + x$ means 'x is the index of a type name;' $l + x$ means 'x is the index of a literal;' $n + x$ means 'it is time to finish the subtree, and x is the node index of its root vertex.' Using the notation $<a : x, b : y, c : z \ldots >$, which indicates a subtree with root a and sons $b, c, \ldots$ having decorations x, y and z, this builds a subtree < `dt_def, fwmType, baseline, ... polarLinear` > (which has no decorations when it is built).

Here is a little bit of the (52 line) type definition for a `forwardModel` specification:

```
begin, s+s_forwardModel, &
       begin, f+f_integrationGrid, s+s_vGrid, n+n_field_spec, &
       begin, f+f_type, t+t_fwmType, nr+n_field_type, &
              nd+n_spec_def, &
```

This indicates that if the optional field named `integrationGrid` appears it must be the label of a `vGrid` specification, the field named `type` is required[§] and must have a value that is a literal of the enumeration type `fwmType`, and duplicate fields[¶] are not allowed. This builds the subtree < `spec_def:d < field_spec, integrationGrid, vGrid` >$\cdots <$ `field_type:r, type, vGridType` >> where the `d` and `r` decorations mean 'no duplicates' and 'required', respectively.

ACKNOWLEDGEMENTS

The author learned how to construct a program as a language interpreter by attending a class on compiler technology, given by Frank DeRemer and Tom Pennello as part of the Summer Institute for Computer Science at the University of California, Santa Cruz, in 1983. Unfortunately, the text for that class, *Compiler Construction by Hand and by Tool*, which includes description of the tree-based type-checking technique, was a manuscript, which has not been published. The author subsequently used the framework learned in that class, cast in Pascal, Modula-2, Ada, and Fortran 95, in many projects prior to EOS MLS, and to teach undergraduate-level compiler classes for 14 years. The teaching materials for those classes, which include a framework similar to that used in EOS MLS, are available from the author in Modula-2, Ada 83, and Fortran 95. This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

REFERENCES

1. Waters JW, Froidevaux L, Harwood RS, Jarnot RF, Pickett HM, Read WG, Siegel PH, Cofield RE, Filipiak MJ, Flower DA, Holden JR, Lau HK, Livesey NJ, Manney GL, Pumphrey HC, Santee ML, Wu DL, Cuddy DT, Lay RR, Loo MS, Perun VS, Schwartz MH, Stek PC, Thurstans RP, Boyles MA, Chandra KM, Chavez MC, Chen G-S, Chudasama BV, Dodge R, Fuller RA, Girard MA, Jiang JH, Jiang Y, Knosb BW, LaBelle RC, Lam JC, Lee KA, Miller D, Oswald JE, Pagel NC, Pulala DM, Quintero O, Scaff D, Snyder WV, Tope MC, Wagner PA, Walch MJ. The Earth Observing System Microwave Limb Sounder (EOS MLS) on the Aura satellite. *IEEE Transactions on Geoscience and Remote Sensing Special Issue on the EOS Aura Mission* 2006; **44**(5):1075–1092.

[§] `nr+`x means 'it is time to build a subtree for a required field and the node index of its root node is x'.

[¶] `nd+`x means 'it is time to build a subtree for a specification for which duplicate fields are not allowed and the node index of its root node is x'.

DOI: 10.1002/spe

2. Snyder WV, Wu DL, Read WG, Jiang JH, Wagner PA, Livesey NJ, Schwartz MJ. Processing EOS MLS level-2 data. *Tech Brief NPO 35188*, NASA, March 2006.
3. Johnson SC. YACC: Yet another compiler. *Computer Science Technical Report #32*, Bell Laboratories, Murray Hill, NJ, 1975.
4. Aho AV, Sethi R, Ullman JD. *Compilers. Principles*, *Techniques and Tools*. Addison-Wesley: Reading, MA, 1986.
5. Fischer CN, LeBlanc RJ Jr. *Crafting a Compiler*. Benjamin-Cummings: Menlo-Park, CA, 1988.

DOI: 10.1002/spe