Functional-based modeling
White paper
June 2009

Rational. software

UML for the C programming language.

Bruce Powel Douglass, PhD, IBM

Contents

Executive summary

Unified Modeling Language (UML) has been highly successful in the modeling of software-intensive systems, including systems-oriented models and realtime and embedded designs. The most common implementation for these models has been C++, with the C language in second place. On one hand, this is surprising because the most common implementation language for realtime and embedded systems overall by far is the language C. On the other hand, UML is used almost exclusively for object-oriented systems development, and most realtime and embedded designs are functionally oriented.

There are three strategies for using UML for developing C realtime and embedded applications:

- *Object-oriented modeling*
- *Object-based modeling*
- *Functional-based modeling*

Object-oriented modeling, object-based modeling and functional-based modeling vary in how completely they use the power and richness of UML.

Object-oriented modeling uses the full richness of UML to represent the system and then forward-generates C code. This is straightforward, and the only tricky aspect of this is the mapping of the truly object-oriented features of UML into C code. There are some common idioms that make this transformation simple: Classes become structs containing data elements and pointers to “member” functions; generalization is implemented by the nesting of the structs representing the base classes; polymorphic operations are generated by producing virtual function tables with name-mangled member functions; and so on.

Object-based designs are even simpler because they don’t use the full richness of UML; specifically, they disallow the use of generalization and polymorphism. This results in designs that more directly and obviously map to the C language features but don’t take advantage of all the conceptual tools available within UML.

The last approach, functional-based modeling, is the focus of this paper. There are many reasons developers may want to avail themselves of the power of UML to represent the different aspects of their systems (e.g., functional, behavioral, interactive and structural) and yet eschew the use of object features completely, such as:

- *The developers may wish to represent legacy C code in models without requiring its re-architecture.*
- *The developers may be more comfortable with more traditional C concepts of files, functions and variables than they are with classes, operations and attributes.*
- *The system under development may be safety critical, and safety-critical designs are more difficult to certify under standards like DO-178B, Software Considerations in Airborne Systems and Equipment Certification.*

Highlights

UML can be a powerful tool for developers who do not use object features in their system designs.

The FunctionalC profile uses a subset of UML for the modeling of functionally oriented, C-based systems.

FunctionalC UML profile

A profile is a specialized version of UML that subsets, supersets or extends UML for a particular purpose or vertical market domain. The FunctionalC profile uses a subset of UML for the modeling of functionally oriented, C-based systems. The primary diagram types defined in this profile are detailed in Table 1.

Diagram type	FC diagram	UML basis diagram	Description
Requirements	Use case diagram	Use case diagram	Represents uses of the system with relations to actors
Structure	Build diagram	Component diagram	Shows the set of artifacts constructed from the source files, such as executables and libraries
	Call graph	Class diagram	Shows the calls and their sequences among sets of functions
	File diagram	Class diagram	Shows the set of .c and .h files and their relations, including
	Source code diagram	<none>	Shows the generated source code as editable text
Behavior	Message diagram	Sequence diagram	Shows sequences of calls and events sent among a set of files, including passed parameter values
	State diagram	State diagram	Shows the state machine for files and how their included functions and actions are executed as events (whether synchronous or asynchronous) are received
	Flowchart	Activity diagram	Details the flow of control for a function or use case

Table 1: FunctionalC profile diagrams.

Highlights

Functional development within UML

Developers can program functionally with UML diagrams by using a UML stereotype called a file, which is simply a graphical representation of a source file. This file is capable of containing all the elements that C developers are used to dealing with, including variables, functions, types, etc. The file is added to the diagram and is used to partition the design into elements in much the same way a class is used to partition a program in object-oriented (OO) programming.

A file is a UML stereotype that acts as a graphical representation of a source file and contains elements familiar to C developers, such as variables, functions and types.

The code generated from the model appears very similar to the structural coding styles with which C programmers are familiar. The profile represents .c and .h files that have the same name and couples them together as one element on a diagram called a file. If you do not use the .c and .h file coupling in your code, then we can represent a .c or .h file individually on the diagram. This means that developers do not need to learn how to do OO design, but can just bring the concepts that they have always used into the next level of abstraction, the model. In essence, using the concepts of files, variables and functions in the model enables C developers to graphically describe their program and generate, "what you see is what you get" (WYSIWYG) code from the graphics. In addition, the C developer can simulate the design at the graphical level on the host PC by executing UML model before going to the target to ensure that the behavior and functionality are correct.

Let’s take the example of a timer file. This file could have the responsibility to keep track of time. It could have variables such as minutes and seconds, and perhaps functions such as reset and tick. The reset function could initialize the variables to zero, and the tick function could increment the time by one second. In the FunctionalC profile, we can create a simple file diagram such as the following that shows a single file called “Timer” that has variables minutes and seconds of type integer, as well as public functions tick() and reset().

Timer <File>
+mins : int +secs : int
-tick():void -Reset():void

Figure 1: Structure diagram.

The C code for this file would look just as you would expect a typical C program to look:

```
extern int mins; extern int secs;
/*## operation Reset() */
void Reset();
/*## operation tick() */
void tick();
```

This code looks the same as what a C programmer might write, except it was generated from adding these elements into the diagram above.

Functions in one file can, of course, communicate with functions contained in another file, and they can also contain behavior defined by either a state diagram or a flowchart. In addition, files and objects can both be used in the same model, and files can be converted to objects, if desired. This enables developers who wish to migrate to an OO approach to do it at their own pace and doesn’t force an all-or-nothing switch.

Highlights

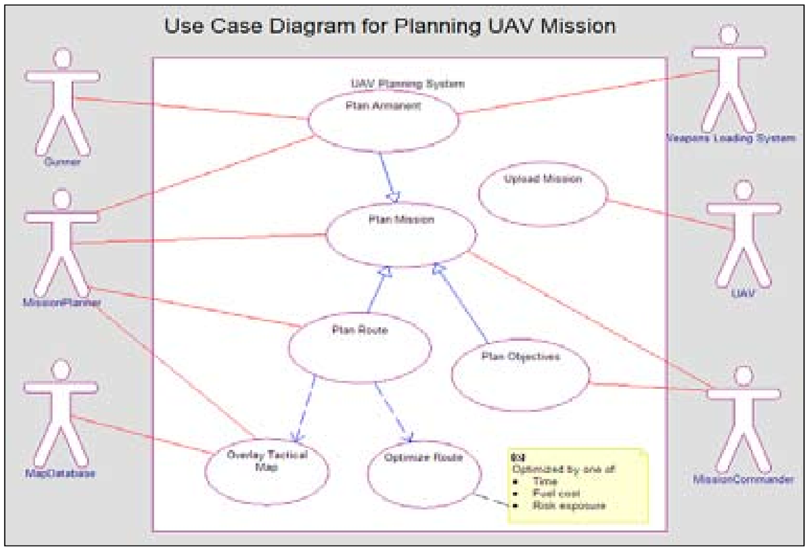


Figure 2: Use case diagram.

Use case diagrams are a powerful way to organize and manage a system's requirements.

FunctionalC diagrams

Use case diagrams are used to render coherent sets of requirements (called use cases) and their relations to external elements (called actors). This is the same as standard UML and is a powerful way to organize and manage a system's requirements.

File diagrams graphically show files, their contents (typically functions, variables and types) and the relations with other files (including source or header file).

Highlights

A file diagram graphically shows a file's contents and its relationship to other files in the system while a build diagram shows the relationships between artifacts.

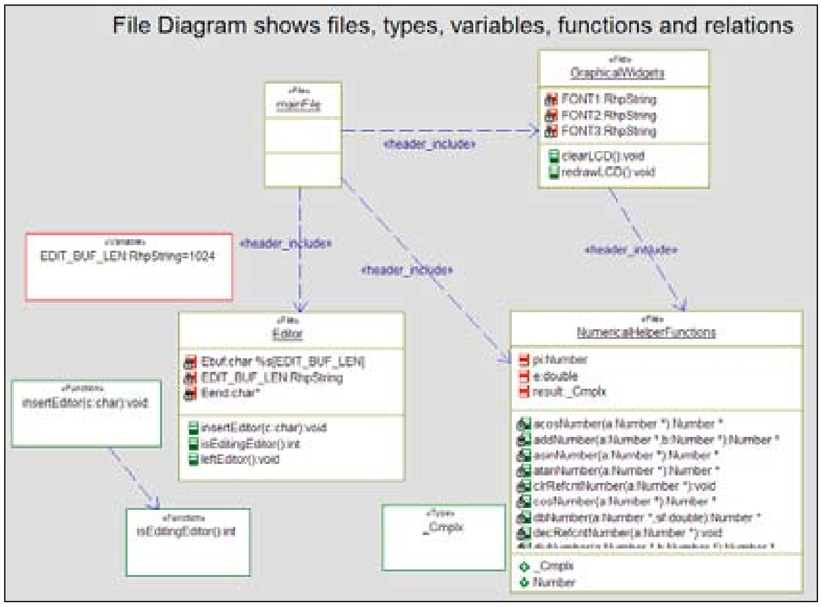

File Diagram shows files, types, variables, functions and relations
«File»
mainFile
«File»
GraphicalWidgets
FONT1:RhpString
FONT2:RhpString
FONT3:RhpString
clearLCD():void
redrawLCD():void
«header_include»
«header_include»
«header_include»
«header_include»
«Variable»
EDIT_BUF_LEN:RhpString=1024
«File»
Editor
Ebuf:char %s[EDIT_BUF_LEN]
EDIT_BUF_LEN:RhpString
Eend:char*
insertEditor(c:char):void
isEditingEditor():int
leftEditor():void
«Function»
insertEditor(c:char):void
«Function»
isEditingEditor():int
«Type»
_Cmplx
«File»
NumericalHelperFunctions
pi:Number
e:double
result:_Cmplx
acosNumber(a:Number *):Number *
addNumber(a:Number *,b:Number *):Number *
asinNumber(a:Number *):Number *
atanNumber(a:Number *):Number *
clrRefcntNumber(a:Number *):void
cosNumber(a:Number *):Number *
dblNumber(a:Number *,sf:double):Number *
decRefcntNumber(a:Number *):void
_Cmplx
Number

Figure 3: File diagram.

The build diagram represents the artifacts constructed via the compilation and link process and how they relate to each other.

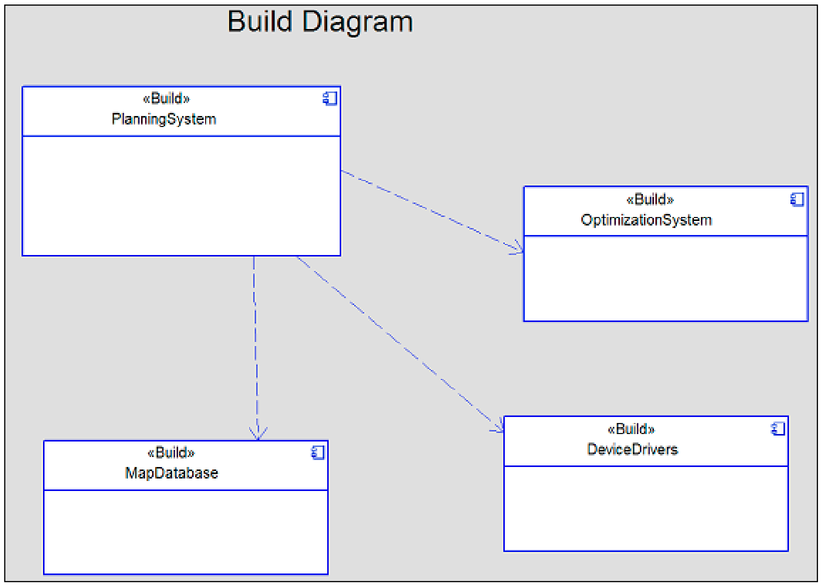


Figure 4: Build diagram.

Highlights

A call graph shows the relationships between functions and variables while a message diagram depicts a specific message-processing scenario.

The call graph shows the relations among functions and variables.

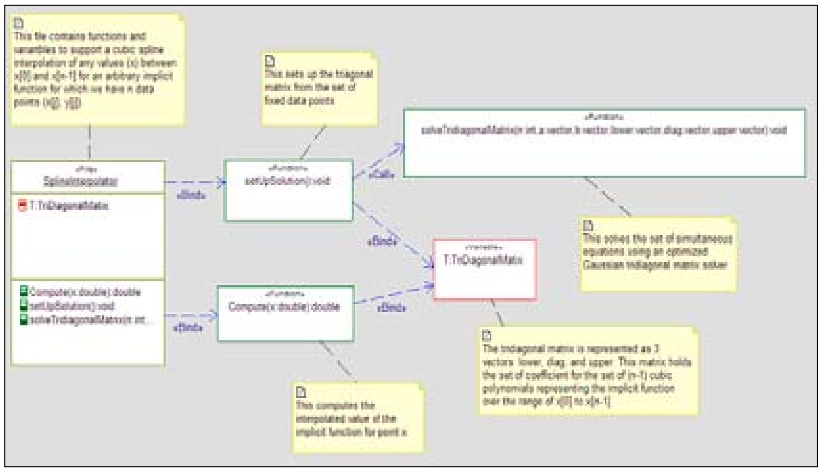

This file contains functions and variables to support a cubic spline interpolation of any values (x) between x[0] and x[n-1] for an arbitrary implicit function for which we have n data points (x[i], y[i])
This sets up the triagonal matrix from the set of fixed data points
«Function»
solveTridiagonalMatrix(n int,a vector,b vector,lower vector,diag vector,upper vector):void
«File»
SplineInterpolator
T:TriDiagonalMatrix
Compute(x double):double
setUpSolution():void
solveTridiagonalMatrix(n int...
«Bind»
«Function»
setUpSolution():void
«Call»
«Bind»
«Variable»
T:TriDiagonalMatrix
This solves the set of simultaneous equations using an optimized Gaussian tridiagonal matrix solver
«Function»
Compute(x double):double
«Bind»
«Bind»
The tridiagonal matrix is represented as 3 vectors: lower, diag, and upper. This matrix holds the set of coefficient for the set of (n-1) cubic polynomials representing the implicit function over the range of x[0] to x[n-1]
This computes the interpolated value of the implicit function for point x

Figure 5: Call graph.

Each message diagram shows a particular scenario in which a specific set of messages (either synchronous or asynchronous) is processed in a particular order. There are usually many message diagrams per use case or per collaboration of files.

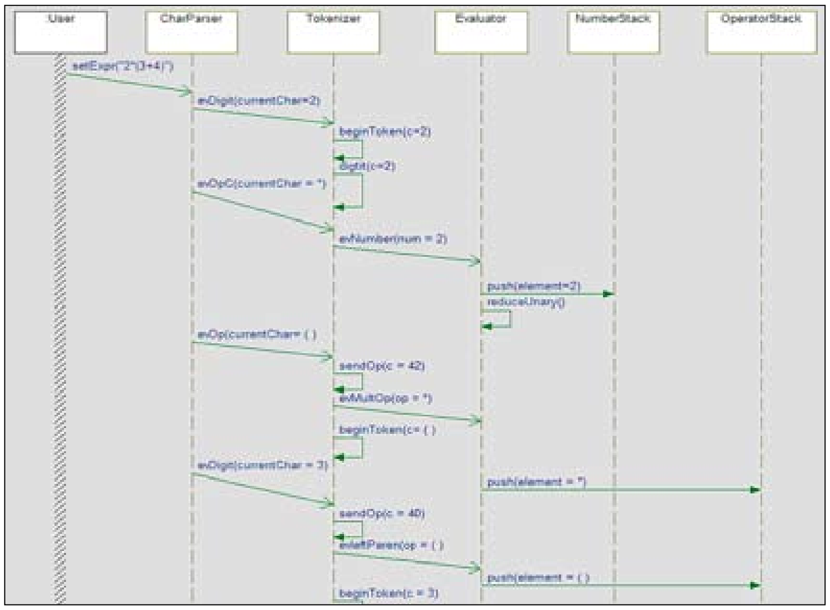


Figure 6: Message diagram.

Highlights

A state diagram shows the set of sets achievable by a file or use case the events it receives and the transactions that result.

A state diagram represents the set of sets achievable by a file (or use case), the events it receives and the transitions that occur as a result of those event receptions.

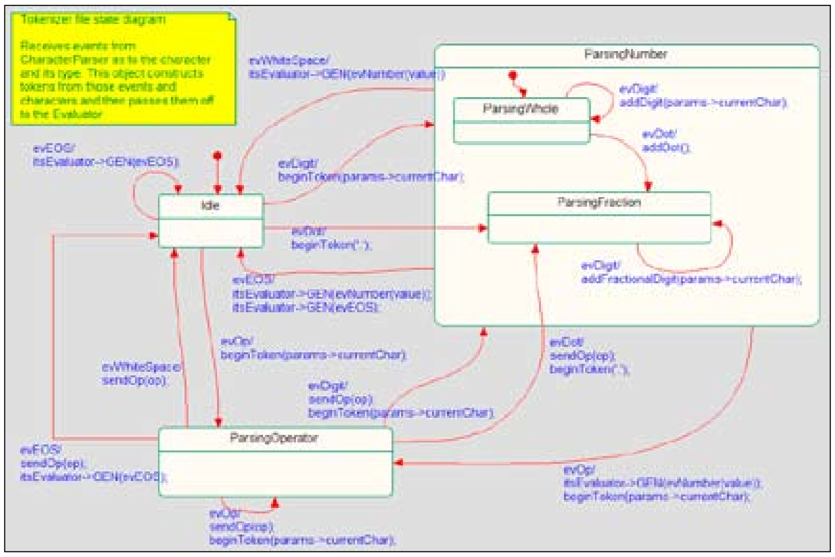
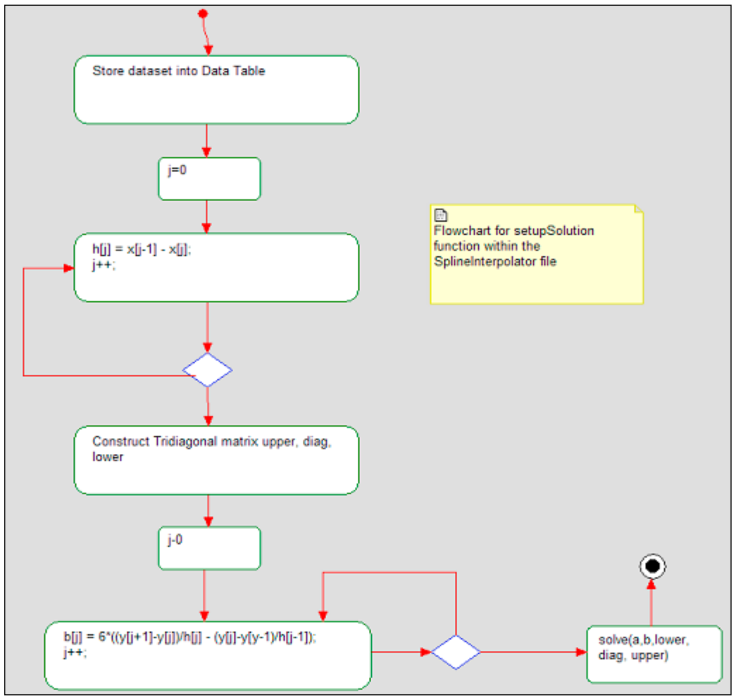

Tokenizer file state diagram
Receives events from CharacterParser as to the character and its type. This object constructs tokens from those events and characters and then passes them off to the Evaluator
ParsingNumber
ParsingWhole
ParsingFraction
Idle
ParsingOperator
evWhiteSpace/ itsEvaluator->GEN(evNumber(value))
evDigit/ addDigit(params->currentChar);
evDot/ addDot();
evEOS/ itsEvaluator->GEN(evEOS);
evDigit/ beginToken(params->currentChar);
evDot/ beginToken(".");
evEOS/ itsEvaluator->GEN(evNumber(value)); itsEvaluator->GEN(evEOS);
evDigit/ addFractionalDigit(params->currentChar);
evOp/ beginToken(params->currentChar);
evWhiteSpace/ sendOp(op);
evDigit/ sendOp(op); beginToken(params->currentChar);
evDot/ sendOp(op); beginToken(".");
evEOS/ sendOp(op); itsEvaluator->GEN(evEOS);
evOp/ itsEvaluator->GEN(evNumber(value)); beginToken(params->currentChar);
evOp/ sendOp(op); beginToken(params->currentChar);

Figure 7: State diagram.

Highlights

Flowcharts may be used to detail the functional flow of control.

A flowchart is used primarily to represent an algorithm or the detailed functional flow of control within a function.


Store dataset into Data Table
j=0
Flowchart for setupSolution function within the SplineInterpolator file
Construct Tridiagonal matrix upper, diag, lower
j-0
solve(a,b,lower, diag, upper)

```
h[j] = x[j-1] - x[j];
j++;
```

```
b[j] = 6*((y[j+1]-y[j])/h[j] - (y[j]-y[y-1)/h[j-1]);
j++;
```

Figure 8: Flowchart.

Highlights

The same diagrams used to describe a model can be used to validate it.

Validating the design using the model

In a typical design, it is never clear that the model is correct until it has been executed. Technology is available today to perform graphical back-animation or simulation, which allows code generated automatically from the model with instrumentation to talk back to the modeling tool. The tool then uses this information to depict the code execution, but in terms of model concepts. This means that the very same diagrams used to describe the model can be used to validate the model. For example, the developer is able to see the value of the variables, see to what each relation is set, see what state each file is in, trace the functions calls between files on a sequence diagram and even step through a flowchart. This animation can be done at any time during a project and allows the programmer to spend more time being highly productive and doing design (the intellectual property), and less time doing the tedious bookkeeping portions of coding. Being able to test and debug a design at both the model and code levels enables the programmer to detect problems early and correct them when they are cheaper and easier to fix.

With realtime and embedded model projects, a significant amount of the code may be generated automatically.

Generating code from the model

Depending on the tool used, C code can be generated directly from the model; all the code that we have seen so far for the timer file can be generated automatically. In fact, the author's experience with realtime and embedded model projects is that on average, a significant amount of the code can be generated automatically. The remaining is code that the programmer writes, such as the bodies for the tick and reset functions. Code can be generated automatically for the dependencies, the association, files, functions, variables, state diagrams, flowcharts and so on. The programmer just needs to specify the functions and actions on the state charts.

Ensuring that the model and the code are always synchronized

In the traditional use of computer-aided software/system engineering (CASE) tools, programmers spend time creating a model and then generate the code. Later the code is modified to get it to work, and nobody ever has the time or energy to update the model. As time goes on, the models get more and more out of sync with the code and become less and less useful. Again, technology is now available to enable any modifications to the code to be "round-tripped" back into the model, helping to ensure that the code and the model are in sync at all times. This is so important during the maintenance phase, as well as when features need to be added to a new version.

Conclusion

The introduction of natural C concepts such as files, functions and variables into UML as a profile helps C developers to receive the benefits of model-driven architecture while thinking and working in the manner to which they are accustomed. Through the process of visualization, it is now possible to incorporate legacy code into the development environment without changing a single line, enabling C developers to reuse their legacy code (IP), either as is or as a starting point. Generation of production-quality structured code directly from the model helps lower risk even further by allowing UML and model-driven architecture (MDA) to fit into existing processes. This helps to further reduce the risk involved in adopting these technologies and can help cut development time immensely. Many companies have already been doing development with UML-based MDA and are finding that they are might be able to reduce the development cycle substantially in some cases.

For more information

To learn more about UML for the C programming language, contact your IBM representative or IBM Business Partner, or visit:

ibm.com/software/rational

RAW14058-USEN-01