DBSF – A NEW DOCUMENTATION-BASED SIMULATION FRAMEWORK FOR AERONAUTIC APPLICATIONS

A. Schöttl, OPS
LFK – Lenkflugkörpersysteme GmbH
Landshuter Str. 26, 85716 Unterschleißheim
alfred.schoettl@mbda-systems.de
Germany

1. SUMMARY

Typical difficulties that arise during the development of algorithms for aerospace applications are:

- the documentation is out of date, misleading or erroneous
- algorithmic errors are discovered at a very late stage of the development process
- the coding of the algorithms is difficult due to different understanding between the algorithms designer and the implementer of the target software

Our approach to cope with this situation is to define a common source in terms of an easily readable standardized document which can be processed automatically. This document serves as algorithms documentation as well as formal algorithms specification by means of an easy-to-learn description language. It contains all necessary information about the algorithms specification as well as its derivation and design history in a readable format (for instance, “math-typed” formulas are allowed). Special emphasis is laid on the interfacing between several simulation components and the simulation variables of the system (including the specification of data types, physical units and interpretation).

The document is processed by a hypothesis based parser. The verification part automatically checks for inconsistencies between data types, physical meanings or physical units’ mismatches. Tests show that the majority of specification errors can be found in that way. The specification is then passed to a C++ code generator which produces code compatible with a comprising simulation library.

2. INTRODUCTION

During the development process of algorithms such as the auto pilot of an aircraft the algorithms developer is faced with a lot of challenges. Generally, classical software development processes and methods (so-called “best practices”) are not designed for algorithms development in aerospace applications. Processes like the V-Model [6] typically assume an existing specification of the algorithm design or assume that the necessary algorithms are easily derivable out of the user specification (e. g. by analysing use cases), which is often the case in graphical user interfaces or data base clients. In the last decade, this type of software has experienced a tremendous growth of importance in the area of e-Commerce applications. Due to this fact modern development processes (e. g. the RUP process [5]) do not see a focus of the software development in the algorithms and its related topics such as derivation or the verification in an appropriate simulation framework, the algorithms coding and the embedding in the target system.

Lacking a standard and facing various requirements (such as coding rules) from different projects, algorithm and simulation developers build up their own individual set of basic tools. A harmonisation of these tools is often not cost efficient in industrial applications. The generation of a reusable piece of software is said to be 3 to 4 times as expensive as a software solution which is only applicable to one project. The situation even gets worse by contradicting obligations of the system software groups, such as the requirement of different coding languages, which may be reasonable from their point of view to ease the verification and maintenance of the resulting system code.

Coding and integration in the system software require a comprehensive set of test data, which is typically generated by a system simulation. The benefit of such a simulation is twofold: The algorithms designers need such a simulation as design framework. Therefore, the simulation is inseparably interconnected with the algorithms and should always been seen together.

Those simulations are mathematically speaking based on solvers of a complex set of ordinary differential or difference equations. The structure of the derivatives is often specified by a set of models (for the environment, kinematics, aerodynamics etc), whose inputs and outputs are linked to each other. During the course of development varying simulations are needed to generate the intended test data, reference data and forecasts of the system behaviour. The necessary reconfiguration of the simulation for the intended use is often more time and resource consuming than the development of the models itself.

Since the standard software development processes do not comprise the development of algorithms as dedicated part, the documentation of algorithms is often not standardized or even not foreseen. Algorithms documentation if often restricted to a specification artefact which serves as input for the implementation and a verification artefact to make sure the implementation fulfills

the specification.

3. VARIOUS APPROACHES

Having in mind this situation, the usage of specialized tools for algorithms development is recommended during the last decade. Code generators shall reduce the effort of recoding the algorithms for the implementation in the system software. Meanwhile there exist well-designed generators (i. e. reliable generators which produce readable code) for the cross-coding between different programming languages (e. g. f2c [2]) and for the coding of specialized algorithms which are designed in a graphical way, especially state machines (processed by tools like Statemate [4]) and block diagrams of control theoretic designs (processed by tools like The MathsWorks Real-Time Workshop [3]). Unfortunately, these generators are efficiently usable only in their dedicated field of application.

A similar situation can be found in the area of simulation. Simulators are only available for special applications. Therefore, typical simulations are built based on specialized software tools (such as Simulink) or coded from scratch as proprietary simulation in a programming language like C++. Using specialized tools often lead to severe restrictions in functionalities like data logging, managing the simulation time, configuration and parameterization. These functionalities are part of the framework and can hardly be adapted to the specific needs. The implementation of those framework features is very extensive; hence most proprietary solutions are also only endowed with a rudimental user interface.

Other tools (such as SCADE [1]) are specialized in the formal and automatic verification of algorithm parts. Those powerful and complex tools are mainly used for the verification of safety critical code. Due to their claim to formally verifying the investigated algorithms completely, these algorithms may only be quite simple and the formal specification of the expected result is complicated.

Another fruitful idea to ease the situation can be found in modern software development processes. They often propose to collect all available information in a commonly used data base (so-called repository). This concept aims in gathering all information belonging to a software module in one place. Unfortunately, adaptations to the special needs of algorithms developers are not contained in these approaches.

4. CONCEPT OF DBSF

Our approach is based on the following concepts: The central role plays a document which contains all development related information including derivation and interfaces. It is similar to the above described block-oriented (control theoretic) tools structured by components and modules. Each component or module section of the document possesses an interface description, which specifies the interaction with the outer world. The connection of inputs of one component/module to outputs of another is part of the “links” section. Each module section is also foreseen to contain chapters for the derivation and the description of the associated algorithms, the according syntax is partially stated below.

The document which is created and maintained by an editor of choice (provided that certain macro programming

FIG 1. DBSF document structure

features are available) eventually using adaptable templates to meet the company’s documentation standards. The output is stored in an XML-format. The DBSF parser checks the format for validity. Due to the interface specifications and some additional notes in the document the parser is also able to perform far reaching consistency and plausibility checks, including the analysis of the contained formulas. If the tests are passed, the DBSF code generator automatically generates DBSF C++ code (typically in form of a dynamic link library (dll)) which contains built-in simulation and monitoring facilities as well as a simple user interface. Without changing the code or the specification it is for example possible to switch on and off the monitoring of each variable in each interface or to reconfigure parameters.

Besides the simple user interface a shared memory mechanism allows the communication to a easy-to-use DBSF graphical user interface with additional possibilities such as the drawing configurable plots online (during the simulation run) and offline (using the data of a previously performed simulation). This mechanism also allows the integration in external simulation frameworks (such as Simulink).

The integration in the system software may then be performed by calling the dll or by directly linking the code to ensure the cross platform compatibility. With respect of the algorithms verification it is worth mentioning that for all described use cases (simple or graphical GUI, embedded code, part of Simulink, etc) and all simulation configurations no recoding or regeneration of code is necessary. If the development platform and target platform are identical, even the compiler has only to be called once.

5. SYNTAX OF THE DOCUMENT

For the sake of shortness it is only possible to describe a subset of the DBSF (type declarations e. g. are not considered here) syntax. Figure 1 shows the fundamental structure of a DBSF document. If two blocks are connected via an arrow, the starting block may be included with several instances in the aiming block.

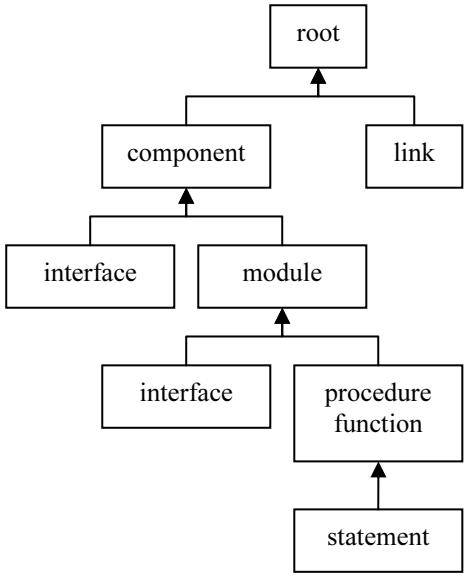

FIG 2. DBSF document structure

The interface section of a module or component describes all variables (including parameters) which may be accessed or modified from outside or shall be managed by the simulation framework. Each variable is assigned a *name*, which very generous naming conventions (greek characters and other special symbols are for instance allowed). The *usage* is indicated by one of the categories input, output, state, parameter or auxiliary. Input and output variables may be linked from and to other modules or components, states are automatically managed (values are saved for reinitialisation, the integration is automatically performed if derivatives are specified etc.), parameters are foreseen for configuration purposes. The *type* of the variable may be an elementary type (such as double, integer, character, Boolean, enumerations etc.) or a composed type (such as structures and arrays). The *category* can be user-defined and categorizes the variables in different classes, such as physical values, measurements, commands etc. Furthermore, the variable *units* define the physical unit of each variable including scalar factors (for scaled values). A *short description* allows a better documentation and provides information which is automatically extracted for the user interface. State variables also contain information about their *integration method*, continuous states are integrated by evaluation of the derivatives, and discrete states are updated manually.

The procedure or function sections specify an arbitrary set of procedures and functions for each module. There are some standard procedures (such as “initial conditions” or “derivatives”) which are called automatically during a simulation run when defined by the user.

The statements of each procedure or function can be subdivided into structural statements such as the classical conditions and loops, calls of other procedures and assignments. Comments are allowed all over the document (and hence, strictly speaking, no statement specific feature). Structural statements may also be rendered graphically by the editor (if it is capable of creating flow diagrams or structograms). Assignments are basically formulas containing an assignment operator (the operator =).

Annex A depicts an excerpt of a simple (and - to fulfil the format regulations of this paper - plainly formatted) DBSF document.

6. FORMULAS

A formula may be written in a math-typed way (it is convenient to use a text processor with integrated formula editor) and coded in MathML format. Readability is improved by allowing operator syntax for functions like sin without following brackets and not insisting on multiplication operators as long as the resulting formula is unambiguous. Matrix operations and constants are also allowed.

Additional features are incorporated, especially designed for algorithm developers. The *derivative* symbols ∂ or dot (as in $\dot{x}$) are used to assign (time)derivatives of states. The *backshift* operator *[! ... !]* allows to access variable values at earlier points of time. Automatic conversion is performed between compatible data types, units or categories. Incompatible variables may be manually converted (as long as reasonable) via a *casting* operator *{ ... }*. The casting operator in prefix notation is used for data type or category casting; unit conversions are specified in postfix notation. The definition of new functions allows argument list- / infix- / prefix- / postfix-notation with definable priorities and applicable data types, categories and units.

The features are implemented in DBSL via a multi-pass hypothesis testing parser which checks various possibilities to find a feasible expression. This enables the users to specify their algorithms in a well readable way, such as the Euler equations

$$\begin{pmatrix}\dot{\varphi}\\ \dot{\theta}\\ \dot{\psi}\end{pmatrix}=\begin{pmatrix}1 & \sin\varphi\tan\theta & \cos\varphi\tan\theta\\ 0 & \cos\varphi & -\sin\varphi\\ 0 & \sin\varphi/\cos\theta & \cos\varphi/\cos\theta\end{pmatrix}\omega_{AB}^{B}$$

where the assignment can be written in matrix form using the classical Greek characters and omitting unnecessary brackets. An expression like

$$\gamma(b+c)\sin ab$$

requires to test several hypotheses (such as γ being a function with the consequence that the brackets are interpreted as function argument followed by an implicit multiplication). In this way, ambiguous expressions can also be detected.

7. RESULTS

The generated code of the DBSF document can be compiled yielding a dll containing all specified modules and components as well as a simple command processor to configure the generated code (“addlog” e. g. adds a variable to the list of variables which will be logged during the next simulation run). Figure 2 depicts the result of directly calling and logging the dll command processor. Figure 3 shows the result using the DBSL graphical user interface, which automatically generates the appropriate commands and calls the dll also without any additional compiling or linking.

```
>getmodules
Rotational_Motion.Turning_Rates
Rotational_Motion.Euler_Equations
Global

>getvars Rotational_Motion.Turning_Rates
M_b
omega_nbb

>addlog Rotational_Motion.Turning Rates_M_b
ok.

>getparams Rotational_Motion.Turning_Rates
I_b

>setparam Rotational_Motion.Turning_Rates
I_b (1 0 0; 0 5 0; 0 0 1)
ok.

>sim
Time: 0.5000
Time: 1.0000
Time: 1.5000
Time: 2.0000
finished.
```

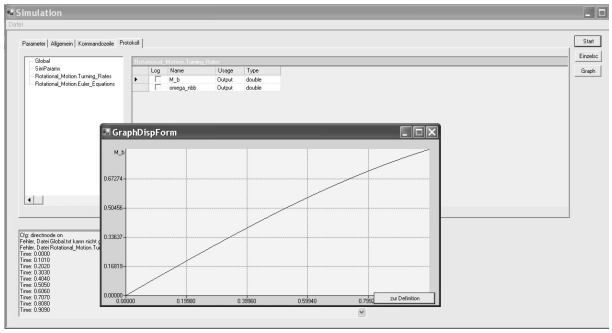

FIG 3. DBSF document structure

8. CONCLUSION

The framework is currently undergoing practical tests to prove to be efficient and comprehensive. First feedbacks show that the users accept the framework due to its concept, which is close to their conventional practices, without any problems. Some extensions especially in the syntax of matrix formulas are currently under investigation.

9. REFERENCES

[1] Jean-Louis Camus; Esterel Technologies: The SCADE Combined Testing Process: Introducing the Compiler Verification Kit, SCADE User Group Seminar 2005, *www.esterel-technologies.com /news-events/events/2005/ ugs.html*

[2] f2c, FORTRAN 77 to C Converter, *ftp://ftp.netlib.org/f2c/*

[3] The MathsWorks Real-Time Workshop, *http://www.mathworks.com/products/rtw*

[4] Model Driven Development Software for Real Time Embedded Systems , I-Logix Software, *www.ilogix.com*

[5] The Rational Unified Process, *en.wikipedia.org/wiki/ Rational_Unified_Process*

[6] The V-Model – process for planning and realization of IT projects – development standard for IT systems of the Federal Republic of Germany, *www.v-model.iabg.de*