TRANSFORMATION FROM GRAPHICAL MODEL REPRESENTATIONS INTO SMP2 MODELS

A. Röhnsch[(1)]; A. Berres [(1)]; O. Maibaum [(2)]; H. Schumann [(2)]

[(1)] German Aerospace Center (DLR)
Simulation and Software Technology
Rutherfordstr. 2
12489 Berlin,
Germany
Email: alexander.roehnsch@dlr.de
Email: axel.berres@dlr.de

(2) German Aerospace Center (DLR)
Simulation and Software Technology
Lilienthalplatz 7
38108 Braunschweig
Germany
Email: olaf.maibaum@dlr.de
Email: holger.schumann@dlr.de

Abstract

The DLR project Virtual Satellite will provide processes for maintaining a repository of SMP2 components. These components' behaviour shall be modelled using a graphical editor rather than directly coding source, as we regard it to be more applicable for system modelling. The model shall then be transformed into an SMP2 model, in order to benefit from platform and simulator independence.

This paper discusses the transformation from different model representations into a corresponding SMP2 model regarding three specific modelling approaches. As the first approach, the tool chain of Simulink with Real-Time Workshop and MOSAIC provides a working transformation from Simulink to SMP2 models. Regarding the transformation, some effort from the modeller is needed. Next, the open object-oriented language Modelica promises advantages like better reusability and versatility. Tools like Dymola provide a graphical editor. But much development is necessary since no transformation into SMP2 models has been developed yet. Last, the SMP2 C++ language mapping and directly coding the component's behaviour provides the technically simplest way. The effort for using each of these approaches will be compared using an exemplary model.

1 INTRODUCTION

In the DLR Virtual Satellite project, SMP2 models are used for simulation. Engineers develop the model behaviour. Reference [1]. It is to manually provide the content for some automatically generated skeleton code. But the currently supported language C++ seems unfitting for direct modelling of concepts of the engineering domain. It is too generic for the task. It doesn't support domain specific concepts and instead burdens the engineer with unrelated concepts and constraints. Engineers may not be adequately trained in the language's specifics. They additionally need to concern themselves with the SMP2 infrastructure. Also, using two sources of code, i.e. the automatically generated and the hand-crafted code, introduces unnecessary difficulties in code handling. Reference [1] therefore proposes the usage of different approaches for behaviour description in the long term.

Dynamic systems can be modelled more intuitively by graphical representations. Modelling environments being specialised in that task seem much more appropriate for behaviour description. Simulink and Dymola are examples of widely adopted tools. They use block diagrams as a direct representation of dynamic systems. Models can be easily created or refined. Fig. 1 shows an example block diagram created with Dymola. Engineers often are already familiar with these programs and the concepts they use. They can therefore easily and quickly define the behaviour of new systems and can generate simulation results. Errors are largely limited to the engineering domain.

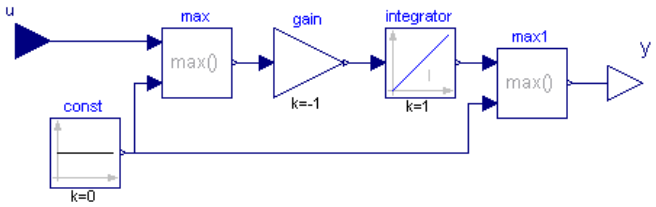

Fig. 1: Example model depicted as block diagram in Dymola.

For the specific modelling and simulation environment Simulink, the Netherlands National Aerospace Laboratory (NLR) developed the program MOSAIC [2], [3]. It provides a transformation from a Simulink model into SMP2 models. The process is described in detail in [3]. Simulink is a widely used commercial tool that is well adapted in the engineering community. The supplementary tool Real-Time Workshop (RTW) generates configurable and reusable simulation code. MOSAIC uses this code for generation of SMP2 models. The current MOSAIC version 7.1 depends on Code of a specific version of Real-Time Workshop [3]. Newer versions are available, leading to issues with Simulink model compatibility.

We describe a general process for transforming the model representation used by these programs into an SMP2 model. In principle, this process could be applied to mostly any graphical modelling tool. The long term goal is to use this process to create a library of reusable, quality assured SMP2 models independent of the tool used for modelling.

2 GENERAL TRANSFORMATION PROCESS

Creating an SMP2 component starts with defining a SMDL Catalogue. In short, the Catalogue describes the model structure. Using the SMP2 Language Mapping, so called boilerplate SMP2 model code (also called wrapper code) can be created from a Catalogue automatically. This model code already takes care of all necessary incantations that can be deduced from the model structure given, e.g. creation and publication of the model's state variables, registration with the simulator, etc. Only the implementation of the model's EntryPoints is left to the user. Regardless of the means used to implement these EntryPoints, compiling the model code yields an SMP2 binary executable.

The model behaviour has yet to be defined by providing the EntryPoint implementation. Instead of directly providing it as hand-coded C++ code, we explain how to use a graphical modelling tool for this purpose. Any graphical or non-graphical modelling tool that creates C/C++ code could be used.

The general process is depicted in Fig. 2. Therein the modelling language Modelica [4] is used as a specific example of a graphical model representation. Given an SMP2 Catalogue that specifies the model structure, a Modelica model template has to be created. This can be automated using model-driven tools. The obtained Modelica model initially only contains structural aspects of the model. An engineer would now define the model's behaviour easily using a graphical modelling tool like Dymola. Doing so results in a complete Modelica model ready to simulate. A Modelica compiler can then generate C++ model code from it.

When generating code from models, a common pattern is to store recurring tasks and code identical to all generated models in a separate stack of code, called runtime library. The model-dependent code parts form the model code. Since the basic execution pattern is the same for all models, it can be found in the runtime library. The model code often consists of several well-known functions with determined semantic. Those functions are called by the runtime library during execution.

It would be nice to just call these model code functions in our own code and let that code be executed by a set of EntryPoints from an SMP2 simulator. However, in order to satisfy the dependencies on which the model code relies, we also would have to reimplement the initialisation and execution patterns of the runtime library. So it seems best to modify the runtime library itself. Given access to the runtime library, its main execution code can be regrouped. The new code would then expose a limited set of basic functions, e.g. "Calculate_Next_Simulation_Step" or "Initialise_Model_Variables". These functions are to be called by one dedicated SMP2 EntryPoint each and scheduled according to their original execution logic. The execution logic from the original runtime library must be understood and implemented in the SMP2 Schedule. Of course different tool's runtime libraries are different from each other and

thus might require different approaches to the choice of EntryPoints. Yet all tools using fixed-step solvers are very likely be divided into an initialise section, one main loop statement and a finalise section.

The model variables have to be published using the SMP2 interface. Typically all relevant variables are accessible through a global data structure. This structure would be declared in the runtime library code, allowing for an easy model-independent publication mechanism. However, instantiation of the variables is model-dependent and would take place in the model code. This indicates that certain information might only be obtained by looking at the model code. This would require parsing each generated model code in some way, and so increasing version dependency and also susceptibility to errors.

Continuing the process in Fig. 2 results in SMP2 compliant code. In that code the actual calculation is done by the model code part from the Modelica compiler, just executed by a set of wrapping EntryPoints. The model variables get published through the SMP2 interface as well. Compiling model and runtime code yields an SMP2 executable.

In order to simulate the executable, SMP2 Assembly and Schedule documents are typically generated as well. They separate model instantiation and execution information from the model structure in the Catalogue. The Assembly is model specific. It describes model instances initial values for their variables. Yet the Assembly's structure can be constructed from the Catalogue, e.g. by a meta-model transformation. The Schedule specifies how EntryPoints are to be scheduled. Its structure can be expected to be the same for all models since all models expose the same EntryPoints defined in the runtime library just before. But depending on the desired simulation settings, each model's schedule may be parameterised differently, for example when choosing different simulation step sizes.

This simple approach of regrouping the execution code and wrap it with EntryPoints works with fixed step sizes. That is fine for some environments, e.g. OpenModelica. But it restricts simulation settings for others, e.g. Simulink. Simulation code that calculates on variable step sizes might have a more complex execution logic. Then a more sophisticated approach than ours will be needed.

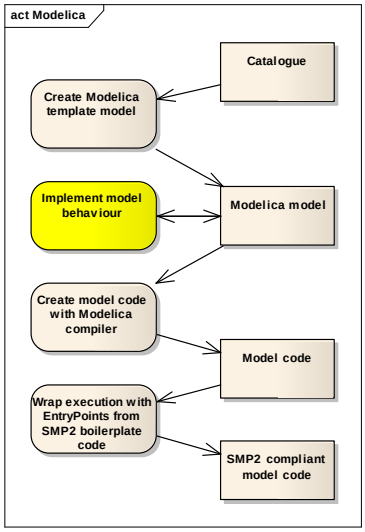

Fig. 2: General transformation process using Modelica as graphical modelling tool.

3 IMPLEMENTATION USING OPENMODELICA

In order to show an example for the general mechanisms of the transformation process, we had to choose a tool working with a specific graphical model representation. Using Simulink for SMP2 model creation is already covered by MOSAIC. We chose Modelica. Modelica is a modelling language for complex physical systems, supporting object-oriented and component-oriented modelling paradigms. So it provides advantages that may be well suited to a real project's demands.

There are a few Modelica compilers available. The most advanced of them is Dymola, completely supporting the Modelica language specification and featuring powerful solving capabilities. But its runtime library source isn't easily

accessible for modification. All source files provided with the distribution rather seem to contribute to model code only. The actual runtime library comes as a binary without source.

This was the resean why we instead chose the modelling and simulation environment OpenModelica. It is distributed as Open Source under the Open Source Modelica Consortium (OSMC) general public licence [6]. It features an own Modelica compiler called OpenModelica Compiler (OMC) [7]. The compiler has good solving capabilities, though still being under development. The OMC translates a Modelica model into C model code. This translation involves substeps like flattening the inheritance structure, resolving inclusions, sorting and optimisation of equations. The functions of this OpenModelica model code are then executed by the OpenModelica runtime library. The Open Source runtime library is accessible for manipulation.

It turned out that execution in the OpenModelica runtime library can safely be separated into the three basic functions *Initialise*, *Calculate_Next_Step* and *Finalise*. Doing so leaves behind a main function that first calls *Initialise*, then *Calculate_Next_Step* in a simple loop statement and in the end calls *Finalise*. Next, this execution logic is implemented with SMP2 mechanisms.

In order to execute the *Calculate_Next_Step* method, an EntryPoint is defined in the Catalogue. The SMP2 Language Mapping automatically declares the EntryPoint in the SMP2 boilerplate model code. Its implementation simply consists of calling *Calculate_Next_Step*. It is scheduled in a regular interval as cyclic simulation time event. Each cycle then calculates a major simulation time step by calling the OpenModelica code function. So the cycle time has to match the simulation step size configured in the OMC compile settings. OpenModelica uses fixed steps for major time steps only. With this the loop statement logic has been implemented. The other two routines have to be called once upon model instantiation and finalisation. So, we didn't implement them as EntryPoints, but let them be called directly from the related SMP2 instanciation/finalisation code.

All model variables are represented in the OMC model code. Their attributes are listed in a global data structure holding all simulation relevant data. No assumptions about the order of the variables can be made. The position of a specific variable has to be determined by searching its name. Variable names are given simply in a "model.submodel.variable" dot notation, allowing for using the identifiers defined in the Modelica model. In order to publish the variables to the SMP2 simulator, their memory locations have to be accessed. This requires proper conversion of the data types used in SMP2 and OpenModelica code. For our simple demonstration of applicability we published only few chosen variables manually. Their values are assigned to separate SMP2 variables with each simulation step. A more general solution has to be developed later.

SMP2 boilerplate code, OpenModelica model code, some self-written interface code, and the modified runtime library are then compiled, producing a binary executable object. SMP2 Assembly and Schedule are currently created manually using the Model Integration Environment (MIE) of the SIMSAT simulator [8]. The creation process has also to be automated in later development.

Exemplary Model Transformation

In order to show an actual transformation example using the above process, a simple model is used. It calculates the remaining charge of a battery. Its input signal *u* is the electric power used by the connected consumers. Its output signal *y* is the electric energy remaining in the battery. Fig. 3 shows the SMP2 Catalogue created for that model specification. At the moment, this is done manually. It contains two SMP2 Fields, representing the input and output signals *u* and *y* as well as the EntryPoint for the simulation loop.

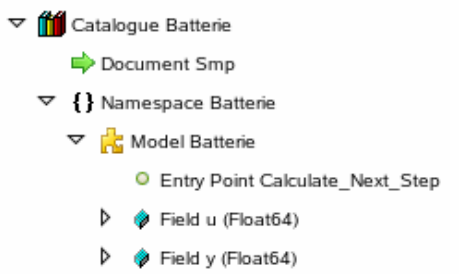

Fig. 3. SMP2 Catalogue of the exemplary battery model.

Let Fig. 4 be an example of the Modelica template model that is to be generated by transformation from the Catalogue document.

```
model Batterie
  Modelica.Blocks.Interfaces.RealInput u;
  Modelica.Blocks.Interfaces.RealOutput y;
end Batterie;
```

Fig. 4. Exemplary Modelica template model.

Let Fig. 1 be an example implementation of this template model. For implementation the graphical modelling tool Dymola was used. The model's underlying Modelica model source code is shown in Fig. 5.

```
model Batterie
  Modelica.Blocks.Interfaces.RealInput u;
  Modelica.Blocks.Interfaces.RealOutput y;
  Modelica.Blocks.Sources.Constant const(k=0);
  Modelica.Blocks.Math.Max max;
  Modelica.Blocks.Math.Gain gain(k=-1);
  Modelica.Blocks.Continuous.Integrator integrator;
  Modelica.Blocks.Math.Max max1;
equation
  connect(u, max.u1);
  connect(const.y, max.u2);
  connect(max.y, gain.u);
  connect(gain.y, integrator.u);
  connect(integrator.y, max1.u1);
  connect(const.y, max1.u2);
  connect(max1.y, y);
end Batterie;
```

Fig. 5: Exemplary implementation for the Modelica model.

For the simulation run, an input signal is specified as shown in Fig. 6. We chose a sine signal since it shows alternating positive and negative power behaviour. Positive power consumption is expected to lead to decrease of the battery energy. On the other hand, negative consumption would lead to energy stagnation. The battery is purposely modelled simple. It is non-rechargeable and does not show interferences upon a negative consumption or upon its emptying.

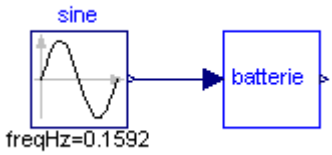

Fig. 6: Environment used for simulating exemplary battery model.

Using the OpenModelica Compiler, the model source code for this simulation setting is generated. We also create the boilerplate SMP2 model code using the SIMSAT Model Integration Environment (MIE). The model source code is compiled together with the modified runtime library, resulting in the SMP2 compliant binary executable of the battery model. Assembly and Schedule are also created manually for the time being. The Assembly simply initialises the model and its variables. The Schedule schedules the one *Calculate_Next_Step* EntryPoint according to the simulation settings. The *Initialise* and *Finalise* functions are called explicitly from SMP2 Initialise and Cleanup code parts. That, too, has to be automated in later development.

Given these steps, the model can now be simulated with an SMP2 simulator. Fig. 7 shows the SIMSAT output of the simulation run. The expected behaviour can be observed. Currently a few problems remain with the OpenModelica output in SIMSAT. One obvious inconsistency is the curve starting at 0 and switching to 10 at the first time step. Apparently this is an initialisation problem. The curve shows that OpenModelica stops processing after 40s, which was

the stop time defined for OMC. In order to provide a longer simulation run, this parameter has to be specified accordingly.

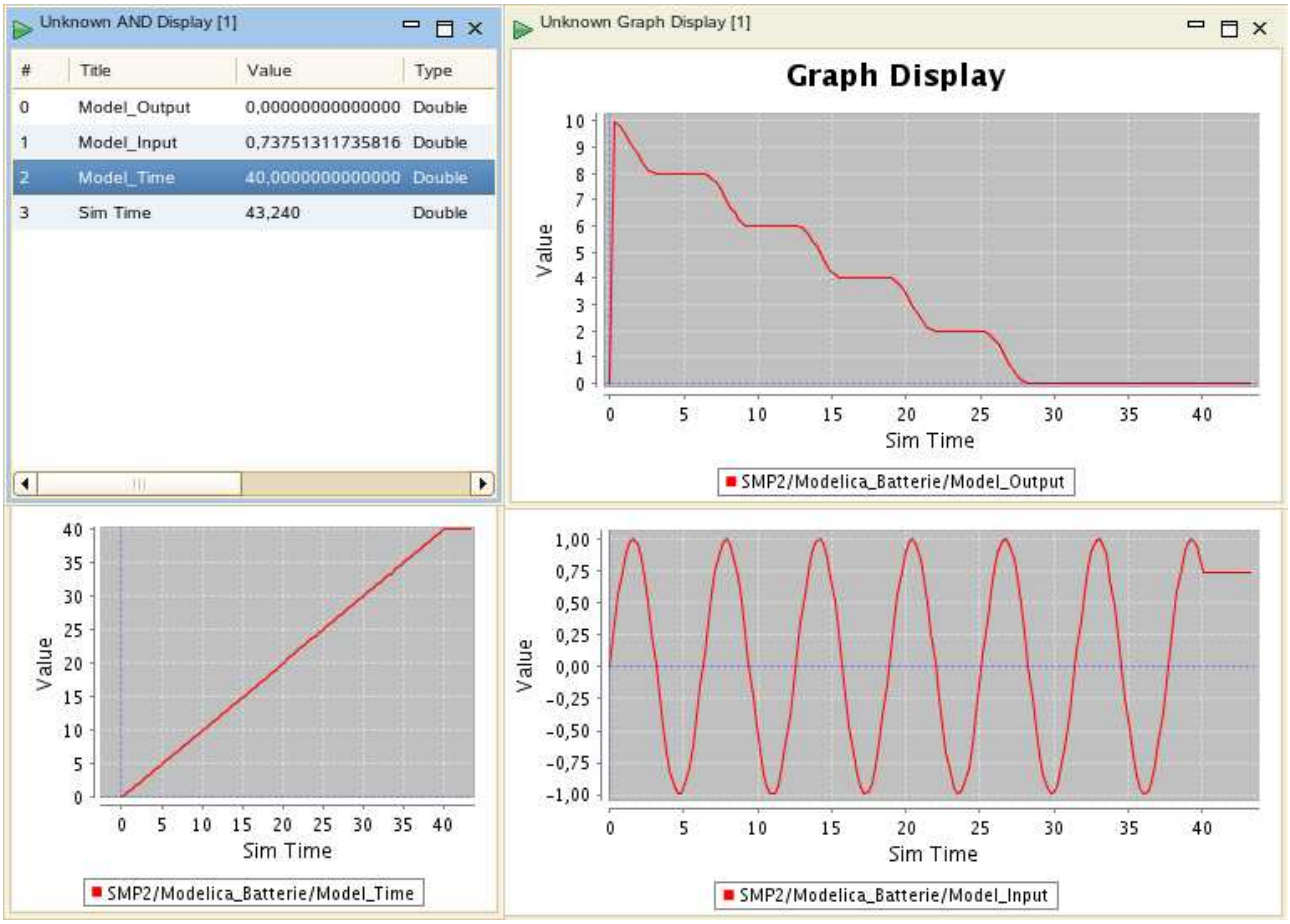

Fig. 7: Simulation of exemplary model in SIMSAT

4 COMPARISON

Ease of modelling is now compared for the standard C++ approach and the graphical modelling approach in general. Concerning two graphical modelling approaches it is then surveyed what steps need be taken for their automated employment as needed in DLR's Virtual Satellite project. These approaches are using MOSAIC with Real-Time Workshop or OpenModelica.

All approaches use the SMP2 C++ Language Mapping for creation of SMP2 boilerplate model code. An existing implementation of the Language Mapping is for example provided by SIMSAT. Directly specifying model behaviour in the created model code using C++ is therefore the technically simplest solution. No further tools for automation or transformation are needed, provided that the engineers are adequately familiar with the C++ language. On the one hand, they then have maximum freedom of choice for their implementation. On the other hand, they are confronted with language concepts not related to the actual engineering domain. More difficulties are introduced by mixing automatic and hand-crafted coding concepts. They have to be dealt with. Thus we regard system modelling using the C++ approach as comparably difficult.

Engineers often are familiar with graphical modelling from using tools like Simulink or Dymola. There, specific engineering paradigms are available for modelling. Thus model errors can be expected to be restricted to the engineering domain. Many simulation environments also feature automated code generation already. The code generation process just has to be modified for SMP2 conformity. Unfortunately, this may introduce restrictions on usable model features and simulation features, resulting in possible modification of preferred modelling approaches. Yet we regard this approach as comparably easy.

Using Simulink as graphical modelling tool includes code generation with the Real-Time Workshop (RTW). This code generation process is automatable with scripts [9]. The subsequent MOSAIC transformation is also automatable through command line scripting. But both RTW and MOSAIC impose model restrictions for their transformation steps to work [3], [9]. Although MOSAIC is compatible with a specific RTW version, a variety of Simulink and RTW versions are available so that already developed models may not be compatible. Thus, existing Simulink models may have to be adapted to work with the release needed by MOSAIC. Another alternative might be an update of MOSAIC itself.

In order to fully automate the process from Simulink model to executable, some development effort is needed for devising a chain of automatic transformation steps.

Modelica is more and more used for modelling in the engineering domain. Being a rather modern, object-oriented language that supports model-driven concepts, it allows for better model separation, model reuse and thus easier development. The OpenModelica Compiler creates simulation code. The process is automatable through command line scripting. Unfortunately no SMP2 transformation has been available yet for OpenModelica code. But we showed that small alterations of the runtime library source can lead to SMP2-compliant code. Although the process has yet to be greatly improved and evaluated. OpenModelica doesn't support the full Modelica standard yet. So it imposes model restrictions, too. Since being under development, it doesn't support the complete Modelica language specification currently. There is development needed for reliably altering the runtime library code in regard to execution controlled by EntryPoints and publication of relevant variables. Also, the generation process for the SMDL documents Assembly and Schedule needs to be developed and the OpenModelica Compiler code generation must be automated.

5 CONCLUSION

Graphical modelling provides for easier and safer behaviour description than manual coding. We identified a general process for transformation of graphical model representations into SMP2 models. We then used OpenModelica as an example of how to implement this general process. Therefore OpenModelica's runtime library had to be manipulated. We generated an SMP2 executable from the Modelica model code. This executable was successfully simulated using a fitting set of Assembly and Schedule documents.

Integrating the fully automated OpenModelica approach requires more development effort than the existing MOSAIC approach. But Modelica offers promising advantages for the modelling process itself. Also it makes a quick adaptation of version changes in the compiler possible.

6 REFERENCES

[1] Simulation Model Portability 2.0 Handbook, 2005, EGOS-SIM-GEN-TN-0099, issue 1, revision 2, 2005.

[2] W.F. Lammen, T. Zwartbol, M. Jansen, A.A. ten Dam, M. Arconi and Q.Wijnands, Automated Model Transfer in Space Applications, Proceedings of SESP 2004 conference, NLR-TP-2004-453, NLR 2004.

[3] W.F. Lammen, J. Moelands, MOSAIC Release 7.1: User Manual, NLR-CR-2006-517, NLR 2006.

[4] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley-IEEE Computer Society Press, 2004.

[5] P. Fritzson; P. Aronsson; A. Pop; H. Lundvall; K. Nystrom; L. Saldamli; D. Broman; A. Sandholm, OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching, Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design, Munich, 2006.

[6] Open Source Modelica Consortium Licence: http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/OSMC/OpenSourceModelicaConsortium-bylaws.pdf (last visited 5 Oct. 2008).

[7] OpenModelica Documentation: http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html#Documentation (last visited 5 Oct. 2008).

[8] Information on SIMSAT: http://www.egos.esa.int/portal/egos-web/products/Simulators/SIMSAT/ (last visited 5 Oct. 2008).

[9] Real-Time Workshop, User's Manual, Version 5, Mathworks Inc., 2002.