SMASS – A Lightweight Satellite Simulation Framework for Concurrent Engineering in Education

M. Berlin [(1)], A. Berres [(1)], K. Briess [(2)], H. Kayal [(3)]

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

[(2)] Technical University Berlin (TU Berlin)
Institute of Aeronautics and Astronautics
Marchstr. 12
10587 Berlin
Germany
Email: klaus.briess@ilr.tu-berlin.de

[(3)] University Wuerzburg
Institute of Computer Science
Am Hubland
97074 Wuerzburg
Germany
Email: kayal@informatik.uni-wuerzburg.de

ABSTRACT

The “Satellite Mission Analysis and Simulation System” (SMASS) for small satellites developed at the Technical University Berlin (TU Berlin) is mainly used in the education of aerospace students at universities with the focus on understanding the relations between system parameters [1]. The design of a satellite with SMASS is based on the dynamic simulation of entire mission scenarios, which goes beyond the classical approach of designing and simulating uncoupled subsystems with static dependencies. However, after a detailed analysis of SMASS, it turned out that it is highly recommended to re-engineer SMASS to make it flexible for future extensions. The goal was to develop a new highly modular structure.

After an introduction, this paper presents the basic layer structure of the new modular SMASS framework. In the next sections, the different layers are explained. This includes their functions and the connections between them. In the end, an example explains the new feature of comparing different satellite configurations using only one simulation run.

INTRODUCTION

SMASS has a server/client structure. The server module hosts the simulation kernel. The clients calculate the space environment and the satellite subsystems (see Fig. 1). All clients can run on different systems like laptops, desktop computers (see Fig. 2), microcontrollers and other hardware components. Consequently, clients for e.g. demanding calculations with high accuracy can be distributed and run on their own machine. But clients can also be replaced by hardware for hardware in the loop (HIL) simulations.

Server and Clients are designed as independent applications. It is possible to start the clients without connecting them to the server. However, this is only useful for standalone tests. To simulate a satellite with several subsystems, the server has to be started first. Afterwards, the concerning clients are started and connected with the server. The connection between the server and the clients is implemented as a TCP/IP connection. Via this connection ASCII messages are exchanged, which transport configuration data, calculation parameter, and results of simulation steps.

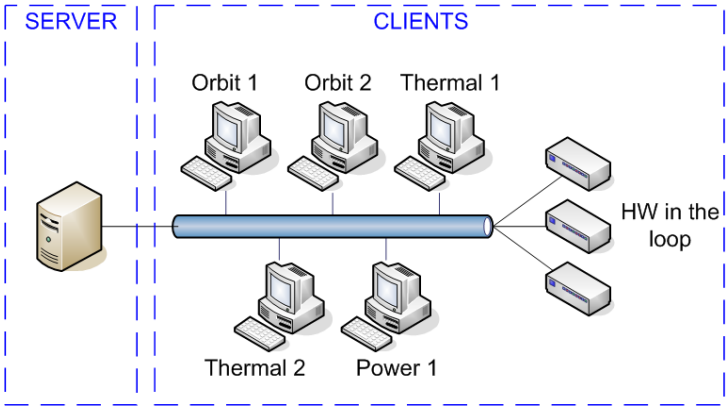

Fig. 1. Connection of simulation modules [1]


Fig. 2. Working environment for the simulation system [1]

Both, clients and server, were developed by aerospace students as part of either their "student research projects" or their diploma thesis work since 2004. At first, the simulation clients for orbit [2], magnetic field [3], thermal [4], and power [5] calculation of a satellite were developed. In 2007, a server connected the simulation clients for the first time [6]. Due to this development order, mutually connected inputs and outputs of different clients had to be converted in an intricate manner by the server developer.

Beyond that, each client has its own communication implementation [7]. This approach is error-prone. It assumes that the students have the same level of knowledge regarding the communication and its software implementation. But in fact, the skills of students are quiet different. In the worst case, these diverse abilities of the clients' developers might lead to clients which can not communicate with the server.

Because students are also in charge of further application development for subsystem calculations, it is advantageous to separate the subsystem calculations from the simulation infrastructure, which comprises the simulation model, business logic, communication, data handling, and user interface.

With that in mind, the SMASS architecture has been re-engineered by joint efforts of the TU Berlin and the German Aerospace Center (DLR) to make it highly modular. A more service orientated framework for the future development of SMASS replaces the old approach. In this context, the term service means offering a calculation of a simulation step in a specific domain like orbit or power.

NEW FRAMEWORK

During the analysis of SMASS, five main functionalities have been identified. The software architecture of the new framework separates these functionalities into layers. It provides well-defined interfaces among these layers. These interfaces are named with an "*I*" for interface or an "*A*" for abstract class as first letter and formatted italic in the UML [8] diagram of Fig. 3. Whether an interface or an abstract class is used, depends on the purpose [8]. The five functionalities and the names of the associated layers are listed in Tab. 1. All five layers will be explained in the following sections.

Tab. 1. Functionalities and corresponding layers of SMASS

Functionality	**Layer name**
Simulation execution and management	Simulation
Domain specific calculation of simulation steps	Services
Loading/saving configuration data and results	Data Handling
User Interface	Frontend
Math methods, XML utilities and communication between Server and Clients	Utilities

Some layers are matching typical layers in software development like the Frontend layer for the user interface and the Data Handling layer for data management. Every layer like the Utilities layer may be additionally sub-divided into packages. These are represented by a yellow folder icon in Fig. 3. This assures an optimal degree of modularity even inside the layers and makes additional classification possible. This modular design with the separation into layers and

packages leads to a gain of expandability and maintainability which were the two most important requirements for the new framework.

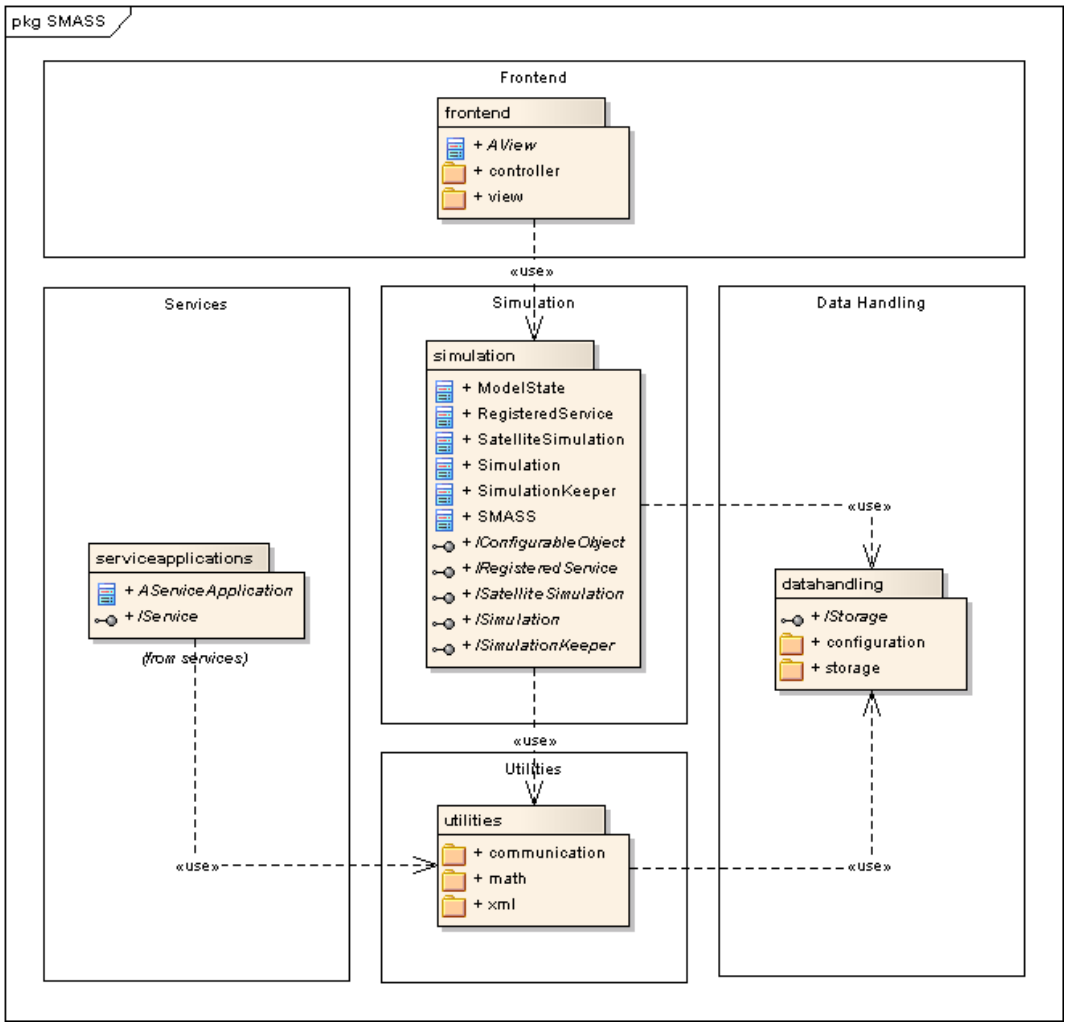

Fig. 3. Layers of new SMASS framework as UML package diagram

The Simulation Layer

The Simulation layer is the main layer of the SMASS framework and is part of the server. It consists of the following components:

- The runtime representation of the simulations,
- the business logic, and
- the enty point.

Business logic implies construction, termination, management, and control of simulations. This is tightly coupled with their runtime representation. For the combination of both, usually the term model is used. The entry point is in general a software stub to start an application. In this case it starts the server application. In Fig. 3, it is denoted as the non-abstract class "SMASS". Non-abstract classes have the same symbol as abstract classes but not italic formatted names without a leading "*A*".

The classes of the model are "SimulationKeeper", "Simulation", "Satellite-Simulation", and "RegisteredService" (cf. Fig. 4). They are ordered hierarchically. The top-most class is the "SimulationKeeper". It manages several simulations, which may run at the same time. Simulations are represented by the class "Simulation" on the next level. A simulation contains at least one satellite. But it may contain multiple satellites, too, which makes an ad hoc

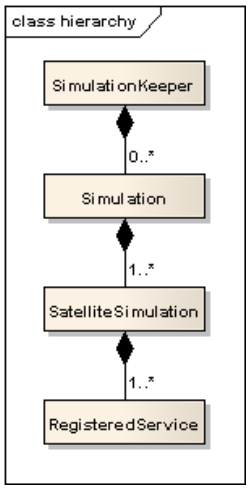

Fig. 4. Model hierarchy as UML class diagram

comparison of different satellite configurations possible. “SatelliteSimulation” represents these satellites. And each of them may have several domains, but at least one. The computation of these domains is performed in services. A service is a part of a client, which have to be registered at the server. After registration, the service is represented on the server side by the class “RegisteredService”. This class is the lowest level in the hierarchical order of the model.

Before starting a simulation, all components of a simulation must be configured. For example, the step size and the initial start time for the simulation and the initial parameter of the services has to be set. The configuration data is loaded by the Simulation layer using the “configuration” package of the Data Handling layer. This dependency between the Simulation layer and the Data Handling layer is also shown in Fig. 3 with the use connection. The arrow points to that part which is used.

After the configuration, a simulation can be started. The Simulation layer sends the request with optional calculation parameters via ASCII messages to the clients. Inside every client, a service calculates the simulation step of its specific domain. Thereafter, each client responds to the request and sends an ASCII message with the results of the computations back to the Simulation layer as part of the server. To send and receive the messages, the Simulation layer, as well as the clients’ services form the Services Layer, uses the “communication” package from the Utilities layer. Again, this dependency is shown in Fig. 3 with the use connection.

Inside of the Simulation layer, interfaces are defined for the access to this layer. The interfaces are independent of the concrete implementations in other layers, which minimizes the dependency between the layers. An example for using these interfaces is the later described Frontend layer. It uses the interfaces of the Simulation layer to provide and display information for the user.

The Services Layer

The Services layer as part of the clients contains the interface for the services and their concrete implementations. For clarity reasons, only the interface of the services including the abstract class “*AServiceApplication*” and the interface “*IService*” is represented in Fig. 3.

The idea of services was developed after the analysis of SMASS’s clients. Each of these old clients contained their own way to implement communication. On the other hand, they had to use the same protocol for the communication in the framework. This is the reason why the communication as well as the interface for services is now implemented in a so-called infrastructure library that is shared by all clients. The new clients are based now on this library and on the Services layer so that one can concentrate on specific service aspects and computation routines for the development of additional clients.

Pfeiff claimed in [6] that a majority of the services must be still developed. This was another reason for the decision to provide an infrastructure library. Beside the communication of the Utilities layer, the library also contains the service interface. That accelerates the client development and standardizes the infrastructure. Due to the bundling of these functionalities in one library, it is easier to maintain the global infrastructure. Furthermore, the service interface leads again to a reduced dependency between layers. The services can now be used in other environments, no matter what kind of implementation, as long as they are able to serve the “*IService*” interface and its methods.

Using the SMASS framework, the service developer has to include the library and implement both parts of the interface for a service. The first part is an entry point, which is implemented by deriving from the abstract class “*AServiceApplication*”. The second part is the interface “*IService*”, which encapsulates three methods. Their functions are:

- calculating a simulation step,
- to configure a service with its initial parameters, and
- returning the current configuration data to be displayed via user interfaces.

Fig. 5 gives an example implementation for the orbit service. It shows the minimal number of two classes needed for a service application. Both classes are located in the middle of Fig. 5 and their names begin with the token “ORB001”. These tokens are a hangover of the old SMASS and have been retained unchanged for familiarity reasons. “ORB” stands for orbit and the number afterwards tells the version of the implementation. A more precise second implementation of the orbit calculation will have the token “ORB002” for example.

The entry point for the service is given by the class “ORB001Application”. The solid arrow symbolizes the derive relation mentioned above between the concrete class “ORB001Application” and the abstract class “*AService-Application*”. The concrete class creates instances of the implementation class “ORB001Implementation” and the message handler represented by a solid line in Fig. 5. “ORB001Implementation” implements the methods of the interface “*IService*”. This relation between an interface and its implementing class is symbolized with a dashed arrow in the UML [8]. The second instance created by the entry point is the message handler for a service. The message handler processes the received ASCII messages and is located in the provided infrastructure library. This connection between the Services layer and the Utilities layer is shown with the use relation in Fig. 3.

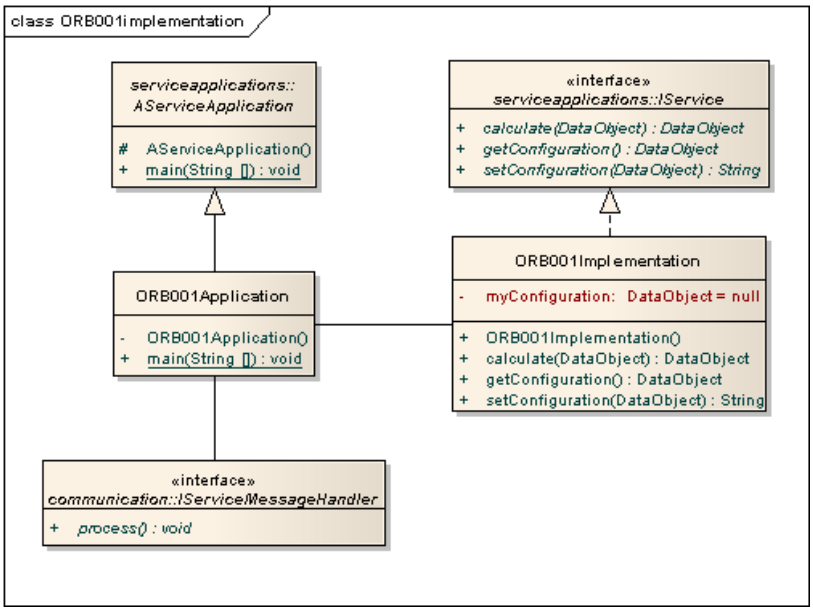

Fig. 5. Example of a service implementation for an orbit application as UML class diagram

The example of Fig. 5 demonstrates how easy now a new service application can be added. The developer just has to create two classes for embedding a service into the SMASS framework. One of these classes contains the functionalities of a service and the other one presents the entry point. Due to the encapsulation of the services, it is not necessary anymore to understand and to implement the internal structure of SMASS or parts of it like the communication.

The Data Handling Layer

To ensure the flexibility and expandability of SMASS, an own layer for data handling was generated. According to Starke [9], this is a common approach in the field of software architecture. The encapsulation reduces the dependencies between layers during their development and the subsequent operational phase of a system. Because of this encapsulation and the development of a suitable interface, it is also possible to exchange the complete data handling. For example, a simple file based persistency can be replaced by a database just by the exchange of the concrete implementation of the Data Handling layer. Hence, the Data Handling layer describes a connector between the data persistency and the SMASS framework.

The Data Handling layer is responsible for the management of configuration data and simulation results. Methods to handle both data types are presented by own packages in this layer (yellow folder symbols in Fig. 3). The “configuration” package is used by the Simulation layer to save and load configuration parameters like step size or start time. The “storage” package is used by the Utilities layer to save or load the results of the simulation steps.

The only valid way to read or write data goes through the Data Handling layer. As the configuration can change during a running simulation, it may be saved for later analysis. Also simulation results can be stored for successive evaluation processes. Thus, the framework allows using results of previous simulation steps as input for other services. This is the prescribed method to exchange values among services. Eventually, this enhanced the old framework that merely aimed at the simulation of uncoupled subsystems to a simulation system for whole satellite mission scenarios.

The Frontend Layer

The Frontend layer includes the user interface for data representation and manipulation. Data to be displayed is queried from the Simulation layer as stated with the use relation in Fig. 3. Beyond that, the Frontend layer is able to change parameters if the user wants to. Thus, several user interaction methods for simulation control and simulation configuration are implemented.

Those two functionalities are further separated inside the layer using the Model-View-Controller (MVC) [10] pattern. The result is the package "view" and the package "controller" (see Fig. 3). In the "view" package, several interfaces for different views are implemented. These views provide optimum user support and display the currently significant data. The "controller" package is in charge of the model manipulation. The third part of the MVC pattern, which is the model, is already included in the Simulation layer.

Advantageous is that the developed user interface can now be exchanged very easily by replacing the classes inside the "view" package. For example, in place of the existing console-based user interface, a web-based user interface might be applied to control the simulation over the Internet. The exchange is possible due to the abstract class "*AView*" and the interfaces inside the "controller" package which are the connections to the Simulation layer. Every added user interface can use the "controller" package without any changes. This new user interface flexibility is a further advantage of the revised SMASS and did not exist before.

The Utilities Layer

The Utilities layer is established for routine operations. This layer is expandable for future operations and contains following functionalities yet:

- communication between clients and server,
- math algorithm used by the services,
- methods for managing and processing data from XML files.

The "xml" package in the Utilities layer (see Fig. 3) contains general methods for reading and writing a XML file, which is the format for configuration files. The "math" package consists of mathematical implementations as approximation methods like Newton or Runge-Kutta. These methods provide a standard set of math operations needed by services. This set will be expanded for further service development.

The most important functionality of the Utilities layer, however, is the already mentioned communication, again embedded in a single package. One part of the communication is responsible to establish a connection between the clients and the server. Right now, this is based on TCP/IP but can be changed to any other protocol by integrating new implementations. After a connection is established, it is used for the registration of client services and for message exchange between the clients and the server.

Also the message handling belongs to the communication. The implementation distinguishes between server and clients. On the server side, the message handling includes the sending of messages to clients and the processing of their responses. Thus, the server is the active part which starts the communication. The clients are the passive part and only react on incoming messages that consist of inquiries for service configuration, of information for the service configuration, and of requests to initiate a computation by services. This is enabled by the interface "*IService*" of the Services layer. The clients' response messages contain the simulation results. These messages are sent back to the server. The server side of the communication stores the results using the Data Handling layer. The relation between the Utilities layer and the Data Handling layer is shown in Fig. 3 again by the use connection.

RESULTS

The results of every simulation step are stored in files by the Data Handling layer which guarantees persistency. Therefore, and as a consequence of the hierarchical model structure of the simulation layer (Fig. 4), multiple satellite configurations can be evaluated later on simultaneously. Fig. 6 shows a comparison of two satellite configurations. In this example, the overall accumulator charge status of two almost identical satellites is supposed to be examined. The satellites differ only in their solar cell configuration. Satellite 1 generates a mean power of 8 W, whereas satellite 2 generates only 6 W in average.

Both satellites start at the identical orbit position in the sun with fully loaded accumulators. In the following Earth eclipse phase, the charge status of both satellites falls to the same local minimum until both satellites enter a sun phase again. As one can see, the second satellite is not able to fully recharge the accumulators during this sun phase. This is a result of the different solar cell configurations.

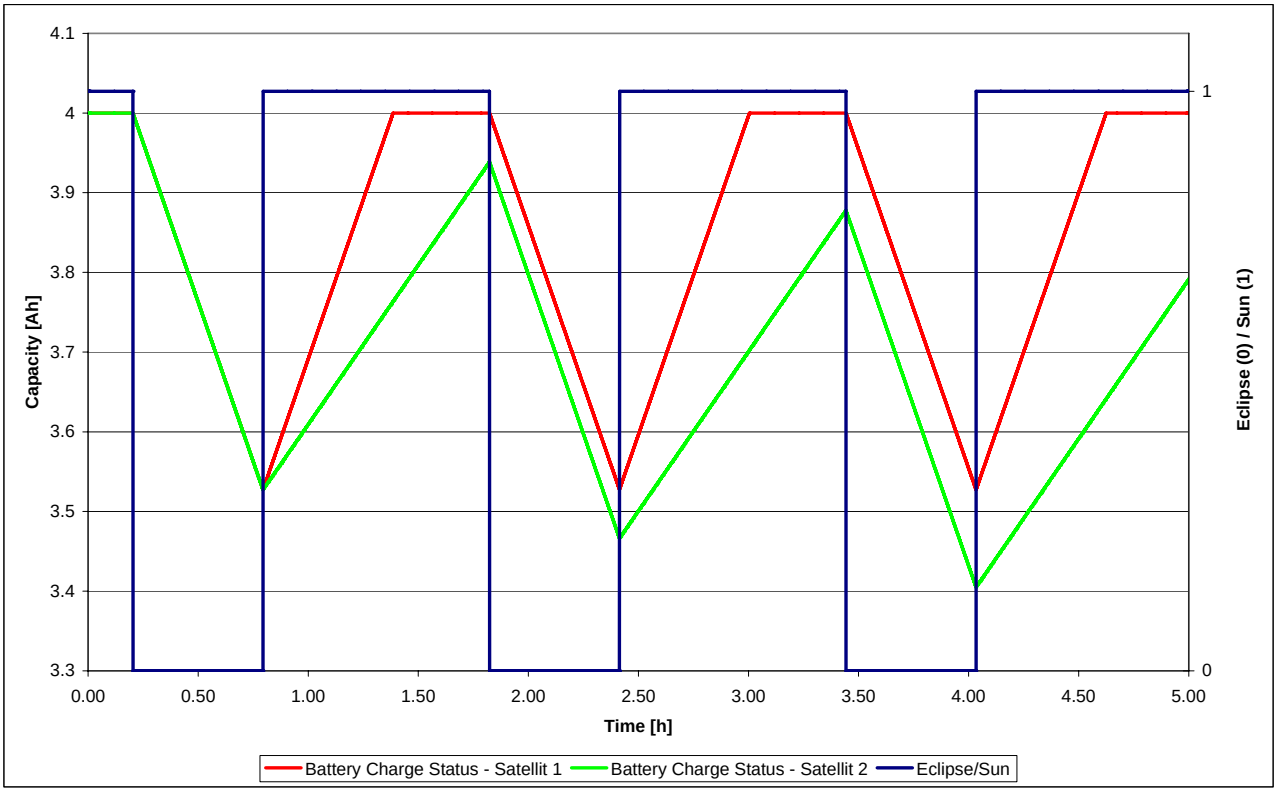

Fig. 6. Comparison of two different solar cell configurations on a satellite on the basis of accumulator charge statuses

This example states the possibility of ad hoc comparison from different satellite configurations using only one simulation run, obtained through the hierarchical model structure. With it the design and engineering process is significantly accelerated.

CONCLUSION

The new framework is highly modular. Therefore, SMASS is maintainable and easily expandable. New applications for subsystem calculations in terms of services can be developed with less effort due to the encapsulation of infrastructure functionalities in a separate library. These functionalities just have to be used by new client services and have not to be implemented again and again. Therefore, the developers, who are mainly students, do not have to understand the internal details of SMASS. They can focus on their own domain such as orbit or thermal calculations, which makes the service development more time effective.

Furthermore, the separation of the services from infrastructure reduces the dependency of services to other framework components. This makes SMASS flexible especially with respect to the infrastructure. Other user interfaces can be integrated in a more convenient way and even the data handling is flexible enough to replace the current file-based input and output by new technologies like a database approach in future work.

The new feature of evaluating multiple satellite configurations within one simulation run is also introduced in this paper. As a result, the time for the development of prototypical satellite designs could be decreased. This is owed to the use of SMASS in education with its 6 months limit for lectures.

REFERENCES

[1] K. Briess, and H. Kayal, "A Mission Analysis and Simulation System for Cost Effective Earth Observation Missions with Micro-, Nano- and Pico-Satellites," Proceedings 55th International Astronautical Congress, IAC-04-IAA-4.11.3.05, Vancouver, Canada, October 2004.

[2] T. Giese, "Aufbau eines Bahnmodells für das Satellitenentwurfszentrum der TU-Berlin," Student research project, TU Berlin, May 2005.

[3] K. Born, "Konzeption und Aufbau des Erdmagnetfeldsimulators für den Luftlagertisch," Student research project, TU Berlin, March 2005.

[4] E. Sandjaya, "Design of a Thermal Model for the Satellite Design Center of TU-Berlin," Student research project, TU Berlin, 2005.

[5] M. Pfeiff, "Simulation der Energieversorgung eines Picosatelliten," Student research project, TU-Berlin, January 2006.

[6] M. Pfeiff, "Entwurf und Implementierung des Steuerungsservers für das Satellitenentwurfszentrum der TU Berlin," Diploma thesis, TU Berlin, October 2007.

[7] M. Berlin, "SMASS – Satellite Mission Analysis and Simulation System – Benutzerhandbuch," Internship report, TU Berlin & DLR, November 2007.

[8] B. Oesterreich, *Developing Software with UML: Object-Oriented Analysis and Design in Practise*, 2nd ed., Addison-Wesley, 2002.

[9] G. Starke, *Effektive Software-Architekturen – Ein praktischer Leitfaden*, Carl Hanser Verlag, 2002.

[10] E. Freeman, E. Freeman, B. Bates, K. Sierra, and M. Loukides, *Head First Design Patterns*, O'Reilly, 2004.