Multi-disciplinary System Engineering and the Compatibility Modeling Language (U)CML

Markus BRANDSTÄTTER
Institute for Astronautics, Technische Universität München
Garching, 85748 Germany

and

Carolin ECKL
Institute for Astronautics, Technische Universität München
Garching, 85748 Germany

ABSTRACT

Over time, technical systems such as automobiles or spacecraft have grown more complex due to the incorporation of increasingly more and different components. The integration of these components, which are frequently designed and constructed within separate departments and companies may lead to malfunctioning systems as their interplay cannot be tested within the earlier phases of development.
This paper introduces compatibility management as one solution to the problems of late component integration. Compatibility management is carried out on a common cross-domain model of the system and therefore allows to test compatibility early on.
We show how compatibility management can be embedded into the phased development of ECSS-M-30A and present the (Unified) Compatibility Modeling Language ((U)CML), which is used for the underlying cross-domain model. A case study demonstrates the application of (U)CML in the development of a small satellite and explains different degrees of compatibility.

Keywords: compatibility management, phased development, multi-disciplinary system model, product life cycle, micro process

1. INTRODUCTION

Technical systems such as automobiles or spacecraft have over time grown more elaborate and complex due to the incorporation of increasingly more and different components. Especially high-tech systems have experienced this transition in the course of which, for example, mechanical and electrical engineering are increasingly being merged with software engineering – for instance in the development of embedded systems. For example, today's cars include up to 80 micro-controllers, which are connected with up to five bus systems that communicate with hardware such as sensors or actuators [3].
Moreover, components are usually developed by independent teams and within separate departments or companies, which frequently leads to malfunctioning systems, the need for major rework, and quite frequently to time and production delays.
The main focus of compatibility management is to assure compatibility of components that exist only as drawings or textual descriptions from the early development phases on to avoid higher costs for their integration in later phases. Therefore compatibility management is centered on defining and providing methods and processes for assuring the compatibility of systems during development, production, and maintenance.

This paper describes a modeling language that allows to test compatibility during the conception and design phases of a product. In a second step the paper describes the phased approach for space systems product development, proposed by the ECSS-M-30A standard. Thirdly, the interplay between compatibility management and phased development, as well as how difficulties with the integration of components affect the product lifecycle is demonstrated. The presented paper concludes with a case study showing the application of the compatibility modeling language in the development process of a small satellite.

2. A NEW LANGUAGE FOR MODELING COMPATIBILITY: (U)CML

Due to the complexity of today's systems, their extensive textual documentation can become very difficult to understand. Therefore key aspects are usually extracted and considered in models of the system. These models provide the basis for understanding and, as [7] states: "models are central objects of scientific communications".
Compatibility cannot be modeled adequately in standard modeling languages like the Unified Modeling Language (UML) or the Systems Modeling Language (SysML), as shown in [2]. That is why the (Unified) Compatibility Modeling Language ((U)CML) has been developed [8].
(U)CML is an object-based modeling language laid out for the design of technical and especially embedded systems (thus incorporating the disciplines computer science, electrical and mechanical engineering). With only a few adaptations, the language can be used for modeling systems of any domain, but this has not been proven so far and is the reason, why the 'Unified' of (U)CML is put in brackets.
The smallest entities of (U)CML are components, which expose their inherent functionality through associated input and output interfaces, which are called plugs. Each plug has a designated direction (input or output) and belongs to exactly one component.
Packages are used as containers to group components. They are organized in a hierarchic tree-like fashion, because a package may contain components as well as other packages, but cannot contain itself. One package (the so called system package) is

therefore the root of the package hierarchy and usually resembles the product to be developed. In order to account for consistency within the model, packages do not possess a functionality of their own, but make interfaces available to their surrounding.
This second type of interface (besides the plugs) is actually called an interface in (U)CML. Within the following sections, only package interfaces (within a (U)CML context) are termed interfaces – plugs are referred to as plugs. Interfaces are used to forward the port information of a plug associated with a component that lies within the package to which the interface is connected. Interfaces may also pass information through their associated package without being consumed (by connecting an input interface of a package to an output interface of the same package).
The key entities for testing compatibility are connections. One connection connects exactly two plugs (plus intermediate package interfaces) and models a channel for the flow of matter, electrical signals or information between the plugs. The introduction of external plugs allows connections to the environment of the system. External plugs are similar to normal plugs, but are connected directly to the interfaces of the root package (i.e. the system).
For the purpose of a bidirectional communication (e.g. a function call), communication variants of plugs, interfaces, connections and external plugs are provided. The graphical representation is identical to their unidirectional counterpart in combination with a reversely directed second plug / interface / connection. Figure 6 shows several elements of a communication between the components "Camera_Mech" and "OBDH" involving the communication plugs 2a and 2b.

All representations of (U)CML entities display two aspects: a graphical shape and an associated list of (compatibility-relevant) attributes, which is called description field in (U)CML (marked by "4" in Figure 6).
The graphical shape shows, with which other entities an entity is associated (plugs are assigned to a certain component, interfaces to a certain package, components and packages to other packages) or with which it communicates (connections between plugs and/or interfaces). Interfaces and plugs on the left side of a component/package indicate an input whereas a position on the right side characterizes an output.
The graphical notation also provides clues as to which discipline the entity belongs and whether an entity failed the test for compatibility by color. The border of a component that is mechanical is marked in green. A plug filled in yellow displays a compatibility warning. A compatibility warning indicates that the marked plug does not fit to its connected counterpart, although this mismatch is not critical.
As mentioned above, every graphical entity has an assigned list of attributes. The values for these attributes are the basis for compatibility tests. For instance, the set of attribute values of one plug are compared with the attribute values of the other plug, which is connected to it by one connection on whether they "match". "Matching" in this context means that given a set of compatibility rules the connection is valid or at least feasible. For example, the compatibility rule "a round building block may be inserted into a square hole" applied in a well-known children's game would cause compatibility warnings in an (U)CML model that models the game. (U)CML is designed to include project-specific and company-specific sets of rules. These two sets of rules may coexist and differ only by the person that administers the rules and by their applicability.
A more detailed explanation of (U)CML and how compatibility is defined is provided in [8].

The practical aspects of modeling compatibility

(U)CML is a useful tool in the development of a system, because its concept enables the translation of models written in other modeling languages like UML or SysML into an (U)CML notation [2]. Thus (U)CML can easily be integrated into an existing modeling landscape. A software editor supporting system design in (U)CML – which is needed for large models – is currently being implemented at the Technische Universität München.
Alternatively, other modeling languages for modeling compatibility in multi-disciplinary systems are SysML in combination with UML, proprietary partial solutions and textual descriptions.
SysML is designed for the creation of integrated models of hard- and software and is supported by various commercial editors. With SysML, the user is able to describe compatibility through the use of comments and specialized expressions formulated in the Object Constraint Language (OCL). But this is a workaround, as comments have been defined for unstructured, informal information. Therefore SysML requires new test routines that filter out relevant information from comments to allow for automated tests for compatibility.
Proprietary modeling environments offer compatibility rules tailored to the company they were designed for. As a drawback, they are designed for only a few systems. Using a proprietary modeling language for a different type of product is usually not feasible and thus requires the costly development of a new modeling environment.
Lastly, textual descriptions provide an almost unlimited number of compatibility rules, but cannot be understood as easily and tested as structured graphical models.

3. THE PROJECT PHASING IN ECSS-M-30A

Aside from tools, system development is guided through a process. A representative development process that is widely used in the engineering of space systems is the standard for project planning [5] by the European Space Agency (ESA). This standard is part of the ECSS (European Cooperation on Space Standardization) set of standards, which define how systems consisting of hard- and/or software are developed within ESA and by its suppliers. The project planning standard ECSS-M-30-A mentioned above addresses phased development of space systems [5]. In the following, the phases are mentioned together with the activities they focus on. The phases are supposed to be traversed in the given order without overlap.

Phase 0 (Mission Analysis/Needs Identification): characterization of the intended mission, needs, operating constraints, possible system concepts

Phase A (Feasibility): finalizing the expression of needs and proposing solutions meeting the perceived needs

Phase B (Preliminary Definition – of project and product): selection of technical concepts for solutions, precise definitions, confirmation of feasibility and determination of operating constraints

Phase C (Detailed Definition – of the product): detailed study of the chosen solution, 'make-or-buy' decisions, initialization of production and verification

Phase D (Production/Ground Qualification Testing): qualified definitions, production for experimental results, integration and verification

Phase E (Utilisation): launch campaign, launch, in-flight acceptance of space elements

Phase F (Disposal): all events from end-of-life until final disposal of the product

Project milestones mark the end of each phase. When such a milestone is reached, a major review is conducted to assure that the product has completed the phase successfully, to determine areas for rework or – in the case of major difficulties in an early phase – abort the project. After phases B to D, the milestones also serve the purpose of setting a controlled baseline (a specified configuration of the product and all its artifacts, which will not be changed anymore).
The ECSS project management standards cover a variety of knowledge areas such as project management, schedule, cost or risk management.
Configuration management is also part of these standards. It specifies which versions of different system artifacts and components have been tested together. Thereby it insures a basic consistency between components.
In addition to that, compatibility management assists in the exchange of consistency information between departments and between models (instead of testing compatibility on components that are already built).

4. THE COMPATIBILITY MANAGEMENT PROCESS WITHIN ECSS-M-30A

This section shows, how compatibility management and the phased development of ECSS-M-30A can work together.
As mentioned above, distributed development and problems with the integration of subsystems are crucial issues when developing technical systems. Within the ECSS-M-30A process, integration takes place in phase D. In phases B and C the components of the system are developed separately. It is evident that an earlier detection of inconsistencies and design errors leads to lower overall costs [10]. Compatibility management insures consistency during distributed development and hence a smooth integration of the system. Therefore, compatibility management should take place in every phase of the development process. During the design phases, compatibility management is based on models, whereas it is based on existing components in the construction phase.
In general, compatibility management is carried out within a process that consists of a sequence of certain steps. These steps are Definition, Identification, Evaluation, Measures, Implementation and Control – called DIEMIC for their first letters [1]. A compatibility manager (CM) is chosen, who controls and coordinates this process with different departments. The CM controls the six DIEMIC steps by focusing on the activities laid out in the following subsections.

Definition
The DIEMIC process starts with a definition of compatibility requirements and an identification of the compatibility-relevant traits of the system.

Identification
When malfunctions, inconsistencies or incompatibilities within the models/components are found, they have to be reported to the CM in an extensive description of the problem. The department then identifies affected components, hierarchy levels and domains together with the CM.
In this step, experts are needed to assure that the observed behavior is a mistake in the system model or an incompatibility.

Evaluation
The evaluation step assesses the criticality of the observed malfunction and requires a decision about the future course of action to restore functionality respectively compatibility of the system.

Measures
Actions occurring in this step are the identification and definition of appropriate measures. Where necessary, other areas of management are brought in (i.e. project management, configuration management, change management and/or problem management).

Implementation
The CM then forwards the suggested counter-measures to affected departments. These document and report back changes to the system and contingently inquire about ambiguities or mistakes.

Control
A last step for the CM is forwarding information about the completeness of the implementation to the initiating department and receiving a confirmation of desired effects of the measures taken. If this confirmation is negative, a new cycle has to be started.

DIEMIC can also be seen as a phased process, but has to be carried out several times throughout the development of a product and does not contain milestones. It can be linked to the phases of the ECSS-M-30A product development process so that this DIEMIC micro process is completed several times within a single phase of the ECSS-M-30A macro process.

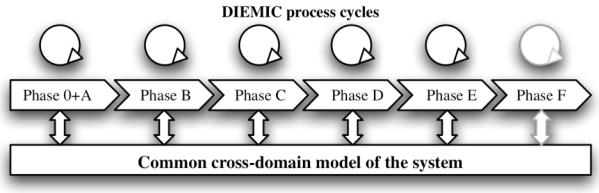

Figure 1: DIEMIC micro process in the context of phased development [2]

During the phases 0 to C, all steps of the micro process are usually carried out, because the most critical decisions are made within these early phases. In phases D through E the definition phase is usually skipped as – in general – no additional compatibility-relevant properties are discovered. The disposal phase F normally does not require compatibility management (indicated through lighter coloring in Figure 1) since products are decommissioned and do not necessitate further design enhancement and correction processes [2].
Besides this, Figure 1 displays a fundament of a common cross-domain model of the system for the phase model and thus for the DIEMIC processes. The thorough utilization of a model of the system common to all departments and disciplines is the main driver for compatibility within the design phases A to C, since it enables the detection of inconsistencies between models at the time the models are created. This may not be an online process, but regular (automated) coordination between the

separate models and the common model already suffices. If only little time elapses between the modeling and integration of the system, the rationale determining the appearance of the subsystem is preserved for evaluating the necessity of recent changes and the proposition of alternative designs.
The ability to detect inconsistencies originates in relating the previously unrelated CAD-models, circuit layouts, UML diagrams etc. When these models are composed, shared components have to be identified and differing perceptions of the component are thus detected.
To sum up the relation between compatibility management and the ECSS standard: it is supposed to be another field of duty alongside project management, change management or problem management (which [6] implements in a micro process in a similar fashion). At the same time, it is strongly linked to these management (sub-) processes and is therefore very pervasive throughout development.
Due to the fact that the development process of a student project is hardly visible, the next section describes the gain from a common cross-domain model of the system in (U)CML.

5. Case Study: CubeSat Move

Currently, a small satellite is being developed within the Institute of Astronautics at the Technische Universität München. Students of different backgrounds are employed to construct the picosatellite MOVE [9]. To be precise, MOVE is designed according to the CubeSat guidelines, which demand a maximum size of 10 x 10 x 10 cm and at most 1 kg weight [4].

Model(s) of the CubeSat

The payloads of MOVE are an optical camera system and an experiment board. Within the following models of the satellite only small sections of the original model(s) are covered. The focus is on the camera payload and the mainboard of the satellite (or in technical terms: the on-board data handling subsystem (OBDH)).
The following figures present MOVE from three different perspectives – Figure 2 is a CAD drawing of the structure of the satellite and the two extracted parts camera and OBDH. The mounting places for both extracted parts are marked by the arrows. It is evident that these two parts have to physically fit into the place that has been reserved for them in order to allow for a functioning satellite. This is an example for compatibility modeling of mechanical parts (within the domain of mechanical engineering).

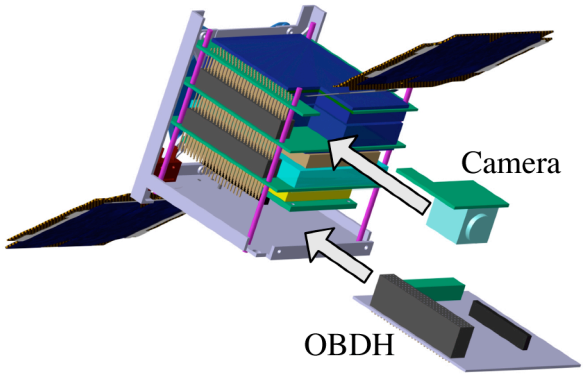

Figure 2: A CAD model of MOVE, its mainboard and camera

Figure 3 is an abstract block diagram displaying the important aspects of the electronic communication between camera payload and OBDH. Less detail is used to model the experiment board and the power supply (EPS). Electronic aspects modeled in the image sensor interfaces of camera and OBDH provide an example for electronic compatibility as the signals sent and those that were expected to be received have to match.

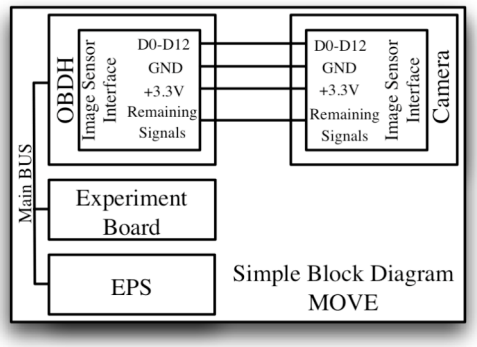

Figure 3: Block diagram of the satellite MOVE

Finally, the following two figures show UML diagrams that model the information exchange and processing related to images and their raw data between camera and OBDH on an object-oriented basis.

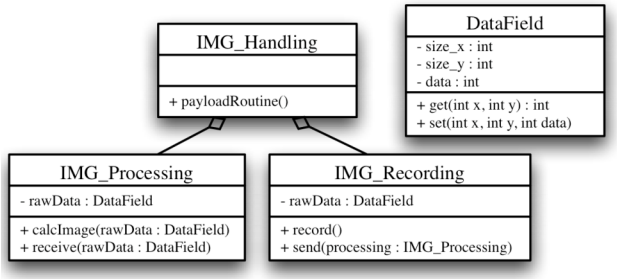

Figure 4: UML class diagram of the image handling process

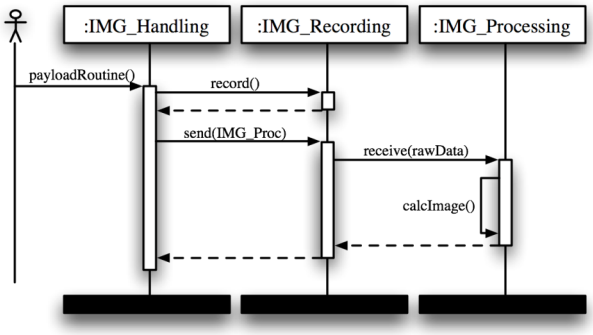

Figure 5: UML sequence diagram of the image handling

All three models are different from each other and represent views from various disciplines onto the same subject. Even here it becomes obvious, that a large amount of effort is needed to connect the camera component to the component OBDH and maintain consistency between the models used. A common cross-domain system model in (U)CML can be constructed from the combination of all three models to facilitate the DIEMIC compatibility management process [2].

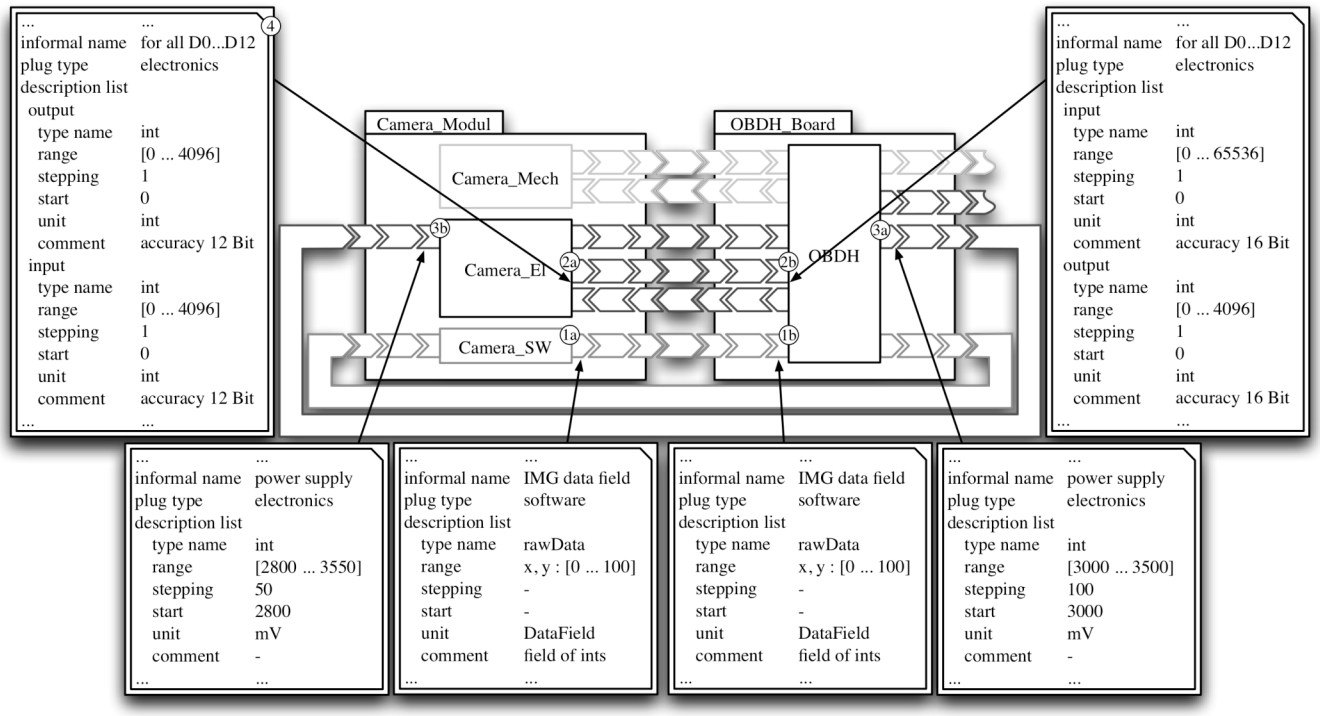

Figure 6: (U)CML diagram of the camera and the OBDH

A possible (U)CML model that incorporates the information of figures 2 to 4 is given in Figure 6. This merged diagram is only the first step towards a true cross-domain model, as links between components that belong to different disciplines cannot be inferred from the original diagrams. Therefore, a second step incorporating connections between components of different domains would greatly improve the utility of the common model. The manual insertion of new connections between electronic and software components requires the collaboration of experts of the involved disciplines and enables automated testing of the connections between different domains [2]. For the sake of clarity and deducibility from the original domain models, these cross-links have been omitted.

The (U)CML diagram in Figure 6 allows system architects (in this case: students) of the involved disciplines to identify their part of responsibility – if modeled correctly as for the camera package and not using one component for more than one represented domain, which is demonstrated in the OBDH component. The engineers have to specify only their (assumptions about) interfaces to components of other domains. Furthermore, as both participants in an exchange of matter, electrical signals or information may reference the same (U)CML connection, communication between engineers from different disciplines is simplified.

Testing compatibility of the (U)CML model

To illustrate the utility of a compatibility test on an (U)CML model, the model drawn in Figure 6 is evaluated based on the values given in the lists of attributes (labeled with "4" in the same (U)CML diagram).

The comparison of plugs 1a and 1b, which constitute the two ends of the connection that describes the handing over of image information, will be described first. This is followed by a test on compatibility of the communication plugs 2a and 2b and finally a compatibility test of the plugs 3a and 3b that model a channel for direct current.

Consistency of the image hand-over: In Figure 6, the description fields connected to the plugs 1a and 1b show the assumptions of both components related to the software information that is exchanged.

Both plugs expect a value of a "rawData" type, which is characterized (in both assumptions) by a range of [0 … 100] in both dimensions and a unit of "DataField". As the plug type is also identical, both attribute lists are the same and compatibility of "Camera_SW" and "OBDH" related to this connection can be assumed.

Of course, this case is very trivial and could be conducted on a simple table as easily. The compatibility examples presented in the following tests will demonstrate that this triviality is not always the case.

Compatibility of the electrical data transfer: This test involves the communication plugs 2a and 2b forming the two endpoints of the electronic communication connection that links up the "Camera_El" to the "OBDH".

This communication is modeled to exchange electrical signals, which enable the transfer of the raw data of the image. As mentioned above, this relationship between electronic and software components is not represented within the (U)CML diagram, because it was not (and could not be) shown in the original domain models.

This communication connection also has a designated direction, meaning that the communication is initiated by one partner – which in this case is the electronic part of the camera.

Looking at the return-direction (from the OBDH to the camera), which is described in the bottom parts of the two description fields, the information of both plugs is identical, because comments do not contribute to compatibility.

In contrast to that, the initiating direction (originating at the camera) does not match completely. The camera sends integers (int) within a range of [0 … 4096], whereas the OBDH could

receive integers ranging from 0 to 65536. This is not a critical difference, since all information sent is preserved. Hence, this would be a (U)CML warning thus indicating that the attribute values of the plugs are not alike, but the connection is feasible. When using a more rigid rule set that does not allow non-equal values for connected plugs, this connection might be marked as erroneous.
Another error can be detected within the power supply and is explained in the following subsection.

Examination of the power supply connection: The diagram displays a third connection between plugs 3a and 3b.
This connection models the channel for the voltage of a directed current between OBDH and camera. Both plugs are tagged as being electronic and provide/consume an amount of mV given as an integer value. These are the only congruities of the description fields belonging to plugs 3a and 3b.
The stepping given within the plugs differs by 50 mV, but the source increases by 100mV and could – by the stepping alone – be consumed by the receiving plug that allows steps of 50mV. Besides the stepping, the two plugs conflict in the range of mV they offer/can receive and in their expected starting values; the output plug 3a delivers a narrower range of voltages than input plug 3b is constructed to accept. This alone would – given a normal ruleset for compatibility – not lead to an error. If, furthermore, the starting voltage of the consuming plug is lower than the lowest deliverable voltage by the source, a compatibility error is certain; a given set of compatibility rules should always evaluate this mismatch as compatibility error, because an implementation of this electrical connection would reveal that a component receiving a higher voltage than it is laid out for will be damaged. For this kind of information, experts are needed, who assert that this damage is certain (as mentioned within the identification step of the DIEMIC process).

In the example discussed within this section, three different types of compatibility conclusions have been discussed including:

- strict compatibility due to identical needs on both sides,
- a compatibility warning because of non-identical, but possibly compatible plugs and
- a compatibility error occurring when differing values for an attribute cause malfunctions within the implemented system

6. CONCLUSIONS

Within this paper, the modeling language (U)CML, its functions and the applicability for space engineering were shown.
We described, how (U)CML could be used within phased development according to ECSS-M-30A. Compatibility management has been introduced as additional knowledge area within space system development and as such is carried out in a micro process of its own. The steps of this DIEMIC micro process have been explained and it was shown how to embed compatibility management into the phases of the ESA standard.
For the cross-domain model that is required to allow for compatibility tests within even early development phases, we chose (U)CML, which meets the requirements for multi-disciplinary modeling of compatibility best.
This paper was concluded with a case study on the development of the CubeSat MOVE. This student project involved mechanical, electrical and software engineering and their respective models. We translated the three models to (U)CML and merged them into a common model. A simple compatibility test was demonstrated on the cross-domain model, which showed three different cases of compatibility – strict compatibility or the exact match of interface descriptions, non-strict compatibility or conformity of interface descriptions by a compatibility rule and a compatibility error.
Currently, an editor for (U)CML is being developed at the Technische Universität München. This editor will be used to facilitate the modeling of large systems and allow for first applications of (U)CML within companies that are concerned with developing (complex) technical systems.

7. REFERENCES

[1] F. Bornemann, S. Wenzel, Managing compatibility throughout the product life cycle of embedded systems – Definition and application of an effective process to control compatibility, INCOSE, 2006.
[2] M. Brandstätter, Modellbasierte Kompatibilitäts-bewertung – Integration von modellbasierter Kompatibilitätsbestimmung in das Systems Engineering Umfeld, PhD Thesis, Technische Universität München, to appear.
[3] M. Broy, "Automotive software and systems engineering", Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings, 2005, pp. 143-149.
[4] CubeSat Community Website, http://cubesat.calpoly.edu/
[5] ESA-ESTEC Requirements and Standard Division, ECSS-M-30A Space project management, Project phasing and planning, Noordwijk, 1996.
[6] R. Haberfellner, P. Nagel, M. Becker, Systems Engineering – Methodik und Praxis, Orell Füssli, 11 edition, 2002.
[7] M. Kayaalp, Modeling and Learning Methods, Technical Report LHNCBC-TR-2004-002, U.S. National Library of Medicine, 2004.
[8] D. Koss, M. Brandstätter, (U)CML – A Modeling Language for Modeling and Testing Compatibility, Software Engineering and Applications (SEA 2007), 2007.
[9] Move project, http://www.move2space.de/home.php
[10] M. Schiffner, Eine objektbasierte Modellierungsmethode für die simultane Systementwicklung, PhD Thesis, Verlag Dr. Hut, 2008