A Reusable Software Architecture for Small Satellite AOCS Systems

Alminde, Lars; Bendtsen, Jan Dimon; Laursen, Karl Kaas

Published in:
Proceedings of Small Satellites Systems and Services conference 2006

Publication date:
2006

Document Version
Publisher's PDF, also known as Version of record

A Reusable Software Architecture for Small Satellite AOCS Systems

Lars Alminde, Karl Kaas Laursen and Jan Dimon Bendtsen
Aalborg University
Department of Electronic Systems, Automation and Control
Fredrik Bajers Vej 7C
9220 Aalborg E
Denmark
Email: {alminde, dimon, karl}@space.aau.dk

***Abstract*— This paper concerns the software architecture called Sophy, which is an abbreviation for Simulation, Observation, and Planning in HYbrid systems. We present a framework that allows execution of hybrid dynamical systems in an on-line distributed computing environment, which includes interaction with both hardware and on-board software.**

Some of the key issues addressed by the framework are automatic translation of mathematical specifications of hybrid systems into executable software entities, management of execution of coupled models in a parallel distributed environment, as well as interaction with external components, hardware and/or software, through generic interfaces.

Sophy is primarily intended as a tool for development of model based reusable software for the control and autonomous functions of satellites and/or satellite clusters.

KEYWORDS

Hybrid systems, Autonomy, Java, Software Architecture, Satellites

I. INTRODUCTION

Small satellite missions tend to be far less costly than their conventional counterparts, which makes it possible to carry out missions that previously were seen as infeasible. Various measures can be considered when attempting to drive the costs down; the satellite hardware itself can be made miniaturized and simplified, off-the-shelf components can be used rather than custom-made ones, and software components can be made generic, allowing to cut down development manhours. This paper describes a software architecture that intends to reduce the time it takes to implement model-based software components such as attitude control, orbit control and fault detection/handling algorithms for small satellites.

The key element is a lightweight application server, named Sophy [Laursen *et al.*, 2005], which is loaded with specifications of hybrid dynamical systems that for instance could represent an attitude controller, a sensor fusion algorithm or a supervisory controller. Each of these models are described in human-readable eXtendable Mark-up Language (XML) documents [Consortium, 2006], which can be reused from mission to mission. The interconnection between the on-line components is specified in another XML file along with parameters specific for the mission.

The Sophy server is written in JAVA, allowing designers to exploit the garbage collecting feature of the Java Virtual Machine [Lindholm and Yellin, 1999], which increases the robustness of the software development process by eliminating the risk of so-called “dangling pointers”. Further, object serialization techniques and networking capabilities are employed to enable the dynamical transfer of components between processing resources (e.g. redundant on-board computers). In fact, the architecture allows the components to be distributed across any persistent network; This means that a Sophy-enabled satellite cluster can share software resources such as e.g. environmental models. From a user perspective, the distribution and networking is completely transparent.

The components are able to interact with sensors and actuators on all platforms in the Sophy network. On each platform the user must extend simple classes to interact with the particular on-board hardware.

Sophy is built around a number of plug-in interfaces that allow users to customize and tailor the framework towards specific needs. This interface also facilitates research, e.g., by making stringent comparison between various algorithms easy.

The design of Sophy was first presented in [Laursen *et al.*, 2005], while this paper describes a second iteration of ideas and interfaces, based on the lessons learned so far.

The paper is organised as follows; First, in Section II, the motivation and objectives of Sophy are explained. Then, Section III presents the formulation of hybrid dynamical systems used in the Sophy framework, whereupon it is shown how these models are mapped to human readable XML files (Section IV). Section V gives an overview of the overall software architecture and a case study (a simulation model of the AAUSAT-II pico-satellite) that involves interaction with Matlab is presented in Section VI. Finally, some conclusions and perspectives are given in Section VII.

II. SOPHY OBJECTIVES

The Sophy framework takes its starting point in three current trends in control engineering systems today, which will be described in this section.

A. Hybrid Systems Modeling

In recent years, hybrid dynamical systems, i.e. systems that exhibit both continuous and discrete dynamics, have been the subject of intense research interest. This paradigm is often well suited when modeling real-life complex systems such as spacecraft, which are describing both continuous variables such as e.g. battery capacity, attitude or propellant masses, as well as discrete variables such as e.g the power-down status of (redundant) subsystems or a commanded mode of operation.

Previous research into hybrid systems theory has focused on hybrid modeling, simulation and verification, see e.g. [Henzinger, 1996]. More recent research has begun to address application of hybrid systems theory to control and estimation problems, see e.g. [Williams and Hofbaur, 2004; Barton and Lee, 2002; Branicky *et al.*, 1998]. In terms of autonomous systems, the potential of hybrid systems used in concert with model-based methods such as optimal nonlinear filtering and model predictive control is huge due to their expressiveness and generality.

So far, however, the research community has not been able to converge on one standard description of hybrid dynamical systems, and one of the objectives of the Sophy project is to develop a specification and corresponding terminology that fits well with spacecraft control applications.

B. Mapping Specifications into Software

Figure 1 (Left) depicts a typical development cycle for model based software. First, requirements are specified and then a simulation model is built that is used to design the algorithms that implement the requirements on the system. The algorithms are evaluated using the simulation model and when found satisfactory, the flight code is written from that basis and verified against the model. Any changes in requirements or models requires a new development cycle in the simulated environment, which then needs to be translated into flight code and tested again etc.

In Sophy, the goal is to bypass the need to write flight code in the design loop, as illustrated in Figure 1 (Right), by automating the process of going from mathematical specifications to an executable software object for simulation and/or online execution. The use of JAVA enables use of the same code both on desktop computers and on flight computers and the plug-in interfaces, together with the distribution capabilities, allow a gradual transition from 100% simulation through hardware-in-the-loop simulation and finally execution on the completed flight system.

Similar features for automatic code generation can also be found in other tools, such as Matlab or MatrixX, but in addition Sophy provides functionality directly supporting hybrid systems theory and distributed computing architectures. These features will be explained in greater depth in the following.

C. Distributed Computing

Traditionally, spacecraft have been controlled from a main on-board computer (possible with redundant backups) — refer to, e.g., [Abildsten and Blanke, 1997]. However, the last decades' rapid advances in digital technology has made ubiquitous computing available even in space. Modern spacecraft typically employ a number of dedicated computers in charge of specific tasks and/or subsystems; one example of small spacecraft employing this strategy is the AAU-Cubesat [Alminde *et al.*, 2004].

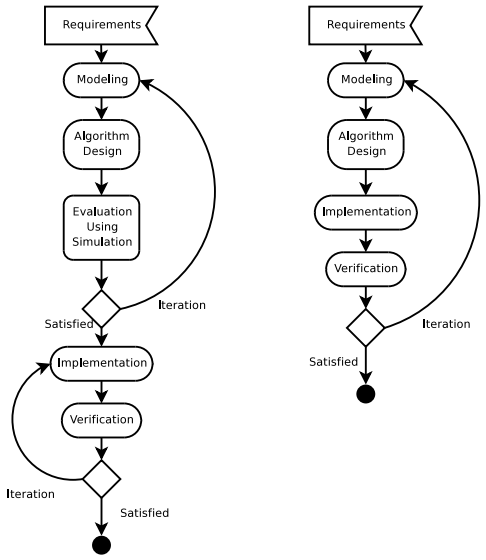

Fig. 1. Workflow for a traditional development cycle (Left) and for development using Sophy (Right)

As documented in [Laursen *et al.*, 2005], Sophy is designed from the bottom up with the understanding of a distributed architecture in mind, allowing to distribute the responsibilities of e.g., flight control systems across a network of nodes and delegate specific tasks to each node.

III. Hybrid Dynamical Systems

This section defines the view of hybrid dynamical systems taken in Sophy, including key definitions of major concepts and practical derivations of the specification. The view that is taken is a control systems-oriented view quite similar to [Branicky *et al.*, 1998], rather than a discrete-events oriented view such as the one found in e.g., [Grasso, 2002].

The specification starts with a very abstract view and then presents some useful restrictions on that. It is the objective of Sophy to support these various views through a highly flexible plug-in-based structure.

A. Abstract Definition of Hybrid System

In the following $\mathbb{R}^n$ will denote the n-dimensional Euclidean space and $\mathbb{Z}^+$ will the positive integers. A hybrid system is an 8-tuple:

$$\mathcal{H} = (Q, X, U, Y, E, \mathcal{F}, \mathcal{G}, \mathcal{T}) \tag{1}$$

in which

- $Q = \{1, 2, \ldots, s\} \subset \mathbb{Z}^+$ is the set of location indexes with cardinal number s

- $X = \{\{x|x \in X_q\}_{q\in Q}|X_q = \mathbb{R}^{n_q}\}$ is the continuous state-space with dimension $n_{q\in Q} \in \mathbb{Z}^+$
- $U = \{\{u|u \in U_q\}_{q\in Q}|U_q = \mathbb{R}^{m_q}\}$ is the continuous input-space with dimension $m_{q\in Q} \in \mathbb{Z}^+$
- $Y = \{\{y|y \in Y_q\}_{q\in Q}|Y_q = \mathbb{R}^{p_q}\}$: is the continuous output-space with dimension $p_{q\in Q} \in \mathbb{Z}^+$
- $E = \left\{e|e \in 2^{\Sigma}\right\}$: is the set of possible input/output event labels, where Σ is an appropriate set of labels
- $\mathcal{F} : Q \times X \times U \rightarrow \dot{X}$ is the forcing function on the continuous state-space
- $\mathcal{G} : Q \times X \times U \rightarrow Y$ is a continuous output map
- $\mathcal{T} : Q \times X \times U \times E \rightarrow Q \times X \times E$ is a transition map

Note how both the continuous ($\mathcal{F}, \mathcal{G}$) and discrete dynamics ($\mathcal{T}$) are allowed to depend on the discrete location (operation mode or similar) as well as the continuous states and inputs, while events affect only the discrete dynamics.

Remark 1 Time is not explicitly included in the definition of the system. However, with no loss of generality the modeler can include an extra state in the continuous map to represent explicit time. ◁

Remark 2 In most practical applications the dimensions of the state-, input-, and output-spaces will not change with different $q \in Q$ (see definition of CDS systems later). ◁

B. Definition of Hybrid Deterministic Systems (HDS)

A HDS imposes the following restrictions on the above definition:

- The maps, $\mathcal{F}, \mathcal{G}$, and $\mathcal{T}$, must be deterministic functions of the state and input
- At any time the total state of the HDS is defined by the triple: $\mathcal{S} = (q \in Q, x \in X_q, u \in U_q)$
- The initial state of a HDS is defined by: $\mathcal{S}_0 = (q_0 \in Q, x_0 \in X_{q_0}, u_0 \in U_{q_0})$
- If the total state is indexed with $q \in Q$, e.g. $\mathcal{S}_q$, it means that the location is fixed, thus: $\mathcal{S}_q = X_q \times U_q$

To define a HDS the initial total state must be included in the definition, and to make the specification of the HDS more convenient the maps $\mathcal{F}$ and $\mathcal{G}$ will be defined as sets of vector fields indexed by $q \in Q$ and the transition map will be broken up into a set of different maps:

$$\mathcal{H}^{HDS} = (Q, X, U, Y, E, \mathcal{F}, \mathcal{G}, \mathcal{T}, \mathcal{S}_0) \tag{2}$$

where

- Q, X, U, Y, E are defined as above
- $\mathcal{F} = \left\{\{f_q\}_{q\in Q} |\{q\} \times X_q \times U_q \rightarrow \dot{X}_q\right\}$ is the set of forcing functions on the continuous state-space
- $\mathcal{G} = \left\{\{g_q\}_{q\in Q} |\{q\} \times X_q \times U_Q \mapsto Y_q\right\}$ is the set of continuous output maps
- $\mathcal{T} = \left\{\{t_r\}_{r\in\{1,..,\rho\}} |Q \times X \times U \times E \rightarrow Q \times X \times E\right\}$ are transition maps indexed from 1 to ρ

In the above, each transition is described as a 4-tuple:

$$\tau_r = \left(j(\mathcal{S}_q), r(\mathcal{S}_q), e_{in} \in 2^{\Sigma}, e_{out} \in 2^{\Sigma}\right) \tag{3}$$

where $j(\mathcal{S}_q) : \mathcal{S}_q \rightarrow \{true, false\}$ is the *transition domain* which triggers the transition when true, $r(\mathcal{S}_q) : \mathcal{S}_q \rightarrow Q \times X$: is an algebraic reset of the state, e_{in} is an input event that causes the transition to trigger and e_{out} is an output event that is emitted when the transition is taken.

In this definition, the use of the location indexed state $\mathcal{S}_q$ rather than $\mathcal{S}$ makes it convenient to group transitions, τ_r, according to source location. For purposes of implementation the transition domain must be specified as a number of logically combined inequalities, for example:

$$j(\mathcal{S}_q) = j_1(\mathcal{S}_q) > 0 \ \wedge \ (j_2(\mathcal{S}_q) > 0 \ \vee j_3(\mathcal{S}_q) > 0) \tag{4}$$

C. Constant Dimension Systems (CDS)

For compositions of hybrid systems, which will be described in the next subsection, the possibility that the dimensions of input, output and state-space changes with each location requires that there are up to $s_1 \times s_2$ different specifications for composing two systems, which can make compositions quite inmanagable.

Therefore, in this paper we will limit the definition of composition to what is called Constant Dimension Systems (CDS). For these system we can use fixed-dimension vectors to represent elements in the spaces, as follows:

- n-dimensional state $x \in X_{q\in Q}\ \forall q \in Q$
- m-dimensional input $u \in U_{q\in Q}\ \forall q \in Q$
- p-dimensional output $y \in Y_{q\in Q}\ \forall q \in Q$

Many practical systems are CDS and it is always possible to embed a HDS in a CDS formulation by looking at the union of spaces over all locations of a system. This may entail that in some locations some inputs or outputs that are unused (kept constant) in the CDS formulation.

Sophy can simulate both HDS and CDS systems, but compositions are always between systems on CDS form.

D. Composition

In the following section we define the parallel composition of two CDS systems, $\mathcal{H}_1^{CDS}$ and $\mathcal{H}_2^{CDS}$, as shown in Figure 2. This operation yields a new hybrid system $\mathcal{H}^{\mathcal{CDS}_3}$. Composing two hybrid systems into one involves two main steps:

- A vector of external inputs $u_{\mathcal{H}_3}$ and outputs $y_{\mathcal{H}_3}$ is selected for the composed system.
- A set of mapping functions that maps the input to the composed system and the output from the two component hybrid systems, $y_{\mathcal{H}_1}$ and $y_{\mathcal{H}_2}$, to the input to the two component hybrid systems, $u_{\mathcal{H}_1}$ and $u_{\mathcal{H}_2}$, and the output from the composed system, is selected for the composed system.

The parallel composition offers the possibility of modeling a complex hybrid system as a number of individual sub-models instead of as a single monolithic hybrid system. The composition is parallel in the sense that the execution of the models happens concurrently.

The composition of two hybrid systems is defined over the domain:

$$||_M : \mathcal{H}_1^{CDS} \times \mathcal{H}_2^{CDS} \rightarrow \mathcal{H}_3^{CDS}, \tag{5}$$

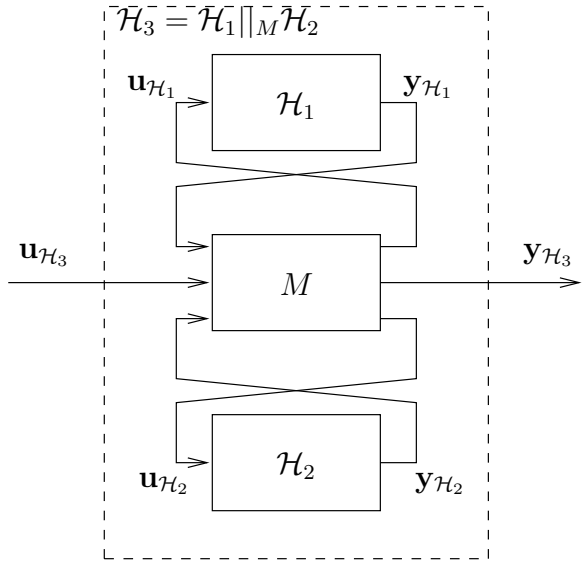

Fig. 2. Composition of two hybrid systems

i.e., composition is an operator that takes two hybrid systems,

$$\mathcal{H}_1^{CDS} = (Q_1,\, X_1,\, U_1,\, Y_1,\, \Sigma_1,\, \mathcal{F}_1,\, \mathcal{G}_1,\, \mathcal{T}_1) \tag{6}$$

$$\mathcal{H}_2^{CDS} = (Q_2,\, X_2,\, U_2,\, Y_2,\, \Sigma_2,\, \mathcal{F}_2,\, \mathcal{G}_2,\, \mathcal{T}_2) \tag{7}$$

and yields a new hybrid system, $\mathcal{H}_3^{CDS}$. It is specified via the matrix mapping:

$$\begin{bmatrix} u_{\mathcal{H}_1} \\ u_{\mathcal{H}_2} \\ y_{\mathcal{H}_3} \end{bmatrix} = M \begin{bmatrix} y_{\mathcal{H}_1} \\ y_{\mathcal{H}_2} \\ u_{\mathcal{H}_3} \end{bmatrix}. \tag{8}$$

with the following restrictions in order to avoid algebraic loops:

- There must only be zeros in the entries mapping between y_1 and u_1
- There must only be zeros in the entries mapping between y_2 and u_2

Note that the dimensions of u_3 (m) and y_3 (p) is implicitly defined by the dimensions of M. It is then immediately concluded that $Q_3 = \{q_3 : q_3 \in Q_1 \times Q_2\}$ must be the location set of the composed hybrid system, $x_3 = [x_1^T \; x_2^T]^T$ is the new continuous state vector (with dimension $n_3 = n_1 + n_2$), and $E_3 = \left\{e | e \in 2^{(\Sigma_1 \cup \Sigma_2)}\right\}$ is the set of possible input/output events

Furthermore, the maps of the composed system can be expressed as follows. Firstly, the composed forcing function is given by:

$$\mathcal{F}\left(q_3, \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}\right) = \begin{bmatrix} \mathcal{F}_1(proj_{Q_1} q_3,\, x_1,\, u_1) \\ \mathcal{F}_2(proj_{Q_2} q_3,\, x_2,\, u_2) \end{bmatrix} \tag{9}$$

where $proj_\alpha \beta$ denotes set projection of α onto the set β. Next, the continuous output map is given by:

$$\mathcal{G}\left(q_3, \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}\right) = \begin{bmatrix} \mathcal{G}_1(proj_{Q_1} q_3,\, x_1,\, u_1) \\ \mathcal{G}_2(pr_{Q_2} q_3,\, x_2,\, u_2) \end{bmatrix} \tag{10}$$

and finally, the transition map is given by

$$\mathcal{T}\left(q_3, \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, e\right) = \begin{bmatrix} \mathcal{T}_1(proj_{Q_1} q_3,\, x_1,\, u_1,\, e) \\ \mathcal{T}_2(proj_{Q_2} q_3,\, x_2,\, u_2,\, e) \end{bmatrix} \tag{11}$$

Remark 3 As seen from the composition, spaces are simply merged and the matrix M distributes information to $\mathcal{H}_1^{CDS}$ and $\mathcal{H}_2^{CDS}$ as appropriate. ◁

Remark 4 As the new $\mathcal{H}_3^{CDS}$ itself is a hybrid CDS, the composition operator is closed. This provides formal justification for the modular design approach outlined in the previous Section. ◁

IV. XML Specifications

As mentioned above, Sophy relies on the XML format for information exchange about models. Specifically, it accepts three kinds of input files:

- Hybrid systems specification
- Specifications of Input/Output connectors
- Specifications of system interconnections

The format of these documents are specified using Document Type Definitions (DTDs). In the following, each document type will be explained.

A. Hybrid System Specification

These documents describe subsystems, such as individual components, in terms of hybrid system formulations. The input files contain XML-encoded representations of the mathematical expressions given in Section III. An XML code snippet is given below, illustrating how a specific location is declared with differential equations, output map, and transitions with transition domains and reset conditions.

```
<location>
 <name>Nominal</name>

 <diffequation state="M">0</diffequation>
 <diffequation state="C">
    (2.7e-4*(M*Pin-20))/V</diffequation>
 <outputmap output="V"> 18+10.6*C
 </outputmap>

 <transitions>
  <transition>
   <name> toPayload</name>
   <domain>C &gt; HFull)</domain>
   <reset>
    <destination> Payload </destination>
    <statereset state="M">0</statereset>
   </reset>
  </transition>
  <transition>
   <name> toSafe</name>
   <domain>C &lt; LCritical)</domain>
   <reset>
    <destination> Safe </destination>
   </reset>
```

```
  </transition>
 </transitions>
</location>
```

In the XML file the declaration of locations is preceeded by specifications of the input and output channels of the system, as per the CDS specification III-C, as well as declaration of numerical constants used in the systems (to simplify expressions in each location). A full example of such a specification is given in the appendix of [Alminde *et al.*, 2006].

B. I/O Connectors

I/O connectors are bridges to the world outside Sophy, e.g. sensors and/or actuators. They can also be used to access data generated by an environmental simulation running in a another software environment such as Matlab before deployment in the proper context.

An I/O connector specification contains information about inputs and outputs to allow composition with other systems by means of the CDS formulation according to the mechanism presented in Section III-D. In addition, it contains the name of a Java class file that acts a a device plug-in. The user most code this small function, as described in Section V.

The I/O connector mechanism is also used for data sinks such as plotting and/or logging.

C. System Interconnections

This XML file contains references to hybrid system specifications and I/O connectors, which are to be loaded and distributed to the network nodes specified in the declaration. Furthermore, it contains information on which input and output channels must be connected between these systems.

V. Sophy Architecture

As outlined in the introduction, Sophy is a framework architecture aimed at implementing advanced autonomy in systems that can be described as hybrid dynamical systems; in general, this concerns complex systems often composed by multiple subsystems. So far, the full architecture of Sophy has been defined and a hybrid simulation component has been implemented and tested. The framework architecture is outlined in Figure 3 and the components are described in greater detail in the following.

A. Input Models

A key architectural point in the framework is that it is *declarative*, in the sense that the user should only be concerned about describing a system and not be concerned about implementing specific controllers and observers for the system at code-level. This is a major break from traditional thinking in automatic control.

To facilitate this, the only human inputs to the framework are hybrid models described in human-readable XML files. On Figure (3) these are indicated as rectangles at the top constituting hybrid models of the different subsystems in the system and a file describing the interconnections between the different subsystems. Any I/O connections to hardware or other software is also declared in XML files. Obviously, as the system models are separated from the simulation architecture in this way, modular design and model reuse is made easier.

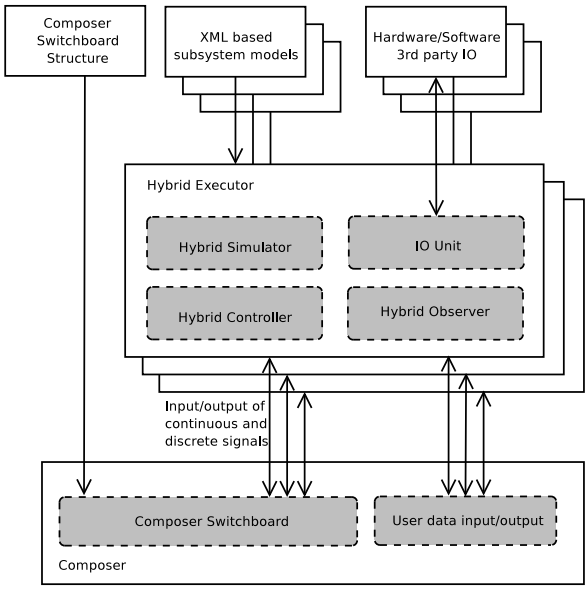

Fig. 3. Overview of the key components in the Sophy framework.

B. Hybrid Executor

The Hybrid Executor (HE) is the architectural element in the framework that is responsible for simulation, observation and control of a single subsystem including managing any I/O interaction with hardware or other software.

A hybrid simulator is implemented and tested in various cases, and a general hybrid observer is scheduled for development within the framework together with a general purpose hybrid controller. General purpose control and observation is envisioned to be implemented using e.g. Unscented Kalman Filtering (UKF) for observation and Model Predictive Control (MPC) for control. Both of these techniques are suited to operate in a declarative environment with a system model as their only input and without the need for comprehensive manual tuning.

The IO Unit also residing in the hybrid executor is a versatile component enabling Sophy to interact with the outside world. This is described in detail in section V-E.

The hybrid executor is not an executable program in itself, but works as a thread spawned by the Sophy Server (see the deployment diagram in Figure 5. When the Sophy Server detects an incoming network connection from a Sophy Composer, it spawns a hybrid executor which is ready to start the internal components required in the current scenario.

C. Composer

The Composer coordinates the information flow between the attached HEs. The switchboard connects data-channels on a subscription basis, meaning that any HE can subscribe to an output of another HE as described in the ”Composed System Structure” XML file. The composer is in this way the central

point of interaction in Sophy, and it is supported by a graphical user interface called the Sophy Desktop Suite.

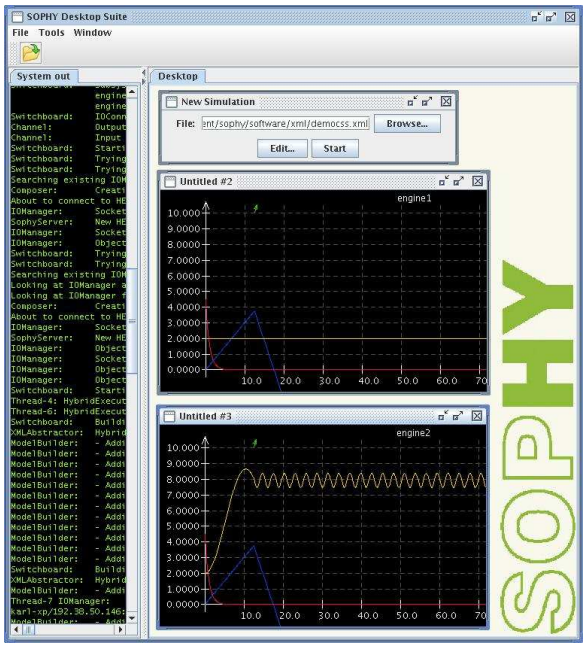

Fig. 4. Screenshot of the graphical user interface, the Sophy Desktop Suite (SDS). SDS encapsulates all the components of Sophy and manages start-up of plug-ins and servers such as the Sophy server which is the host for launching the hybrid executor components on the local platform. The Sophy embedded simulator can be run as a plug-in within the suite along with the PlotViewer visualization tool. Also, a simple XML text editor is integrated into the program

D. Deployment

Sophy is implemented in the Java 1.5 language and all data traffic in the distributed framework nominally uses the TCP/IP protocol nominally; however, if required, a user can write plug-ins that enable Sophy to operate on other network protocols. This is done by extending a simple base class representing a networked connection, the IOManager class.

This allows the components to be distributed as depicted in Figure 5 where a computer is appointed the role of composing the activities carried out on a number servers running the Sophy Server application each hosting one or more Hybrid Executors. When running the Sophy Desktop Suite, all the servers are automatically set up on the local host, but if Sophy servers are running on remote hosts, these may also be exploited when using the suite; addresses for hosts are defined in the Composed System Structure XML file.

A typical simulation scenario involving I/O will start with reading in data from XML files and then distributing the tasks to the desired computational units. The steps are illustrated in Figure 6.

Figure 7 depicts the simulation algorithm in slightly more detail. A simulation is managed by the composer-switchboard, which requests the simulators involved to propagate one time step into the future using the hybrid system models provided during the set-up phase. The simulators do the propagation

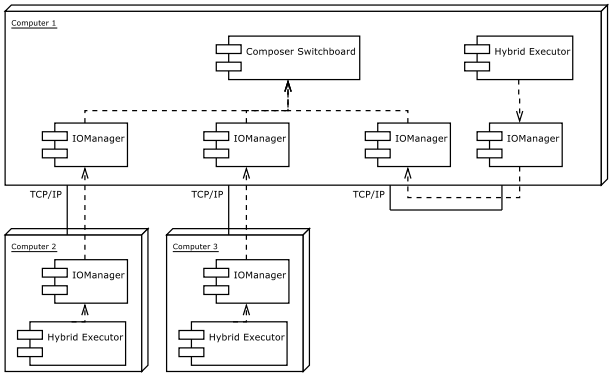

Fig. 5. Deployment of the software components

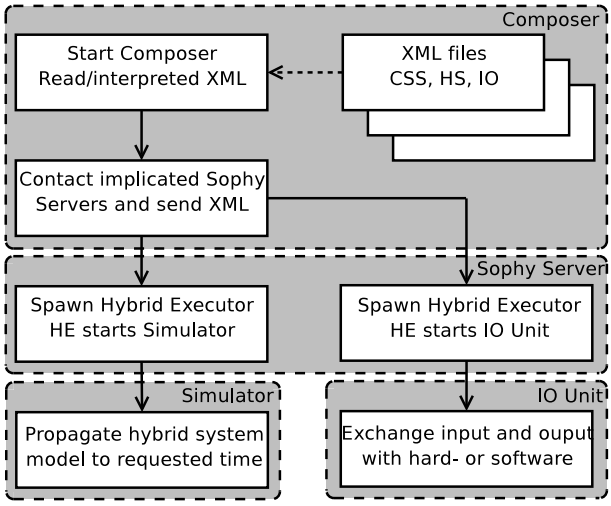

Fig. 6. The deployment sequence of a typical simulation scenario with some I/O in the loop.

and report back to the switchboard with the resulting output signals, and the switchboard then relays outputs to the input of any simulators that have subscribed to other simulator's outputs. The switchboard performs these operations in an infinite loop until the simulation has reached the designated stop time.

E. I/O Infrastructure

The I/O infrastructure of Sophy is one of the features that truly sets it apart from other tools, as it enables interconnection with hardware and third-party software, thus making the framework truly versatile. For example, in the case study presented in Section VI, the I/O infrastructure is employed to build a bridge between Sophy and the commercial Matlab/Simulink package.

The component managing the inputs and output of Sophy is the IO Unit in the Hybrid Executor. The IO Unit has two interfaces to its surroundings: IOManager and IOAdapter. The IOManager is the internal connection to Sophy; as described in the previous subsection, the default IOManager, which uses the TCP/IP protocol, can be overridden to use other network protocols instead.

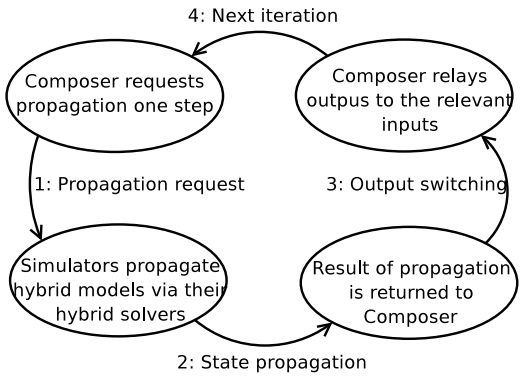

Fig. 7. The overall sequence carried out at each step in a simulation scenario.

The IOAdapter encapsulates a number of implemented drivers to communicate outside Sophy. An example of such a driver is the MatlabAdapter used in the case study. By extending an abstract Java class, the MatlabAdapter allows the IO Unit to send data to and receive data from Matlab, which is then relayed to the Sophy Composer Switchboard. In Matlab, the communication is handled through a Simulink block designed specifically for interconnection with Sophy. Figure 8 gives an overview of the classes comprising the IO Unit package.

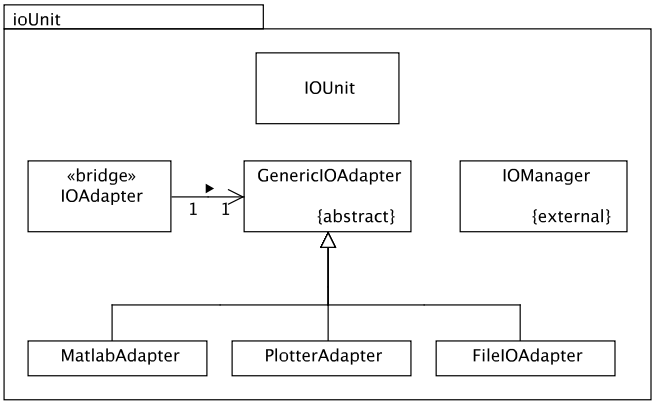

Fig. 8. The Java classes that constitute the IO infrastructure of Sophy. The child classes at the bottom are specializations of the abstract GenericIOAdapter class, which allow Sophy to interact with e.g. Matlab and CSV files during a control or simulation scenario.

VI. AAUSAT-II Case Study

The AAUSAT-II mission is a successor mission to the AAU-Cubesat mission [Alminde *et al.*, 2004], again designed and built by university students. It is built adhering to the cubesat concept specifications, meaning that the dimensions must be only $10 \times 10 \times 10$cm and a mass of one kilogram. An artist's rendition of the satellite is given in figure 9

The satellite builds on technology from AAU-cubesat and has a number of technical goals:

- demonstrate a new type of Gamma ray burst detection instrument

Fig. 9. Artist's rendition of AAUSAT-II in orbit

- demonstrate the use of reaction wheels for three-axis control for such a small platform
- after nominal mission; demonstrate a mechanism to deploy a 40x10cm paddle to provide an extended area for power generation by photo voltaic cells

The satellite will be launched in June 2007 from the Satish Dhawan Space Centre in India to a sun synchronous orbit with an altitude of 600km.

A. Attitude Control System

The attitude control system is based on a combination of magneto-torquers and tiny reaction wheels. A supervisory controller switches between various control modes according to the given conditions at any times. The design of the supervisory control system can be seen on figure 10.

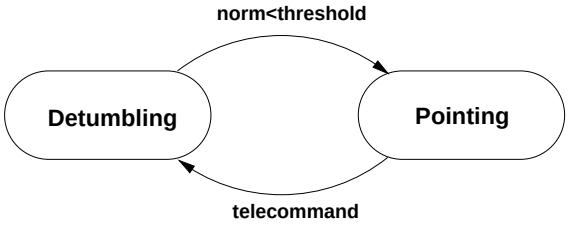

Fig. 10. Supervisory control of AAUSAT-II

In the **Detumbling** location the so called B-dot control law is used to detumble the satellite down to an acceptable momentum for commanding and receiving data from the satellite. The control algorithm only makes use of derivative information about the local magnetic field. Detumbling is a prerequisite to start pointing control in order not to saturate the reaction wheels.

In the **Pointing** location the satellite stabilizes inertially in accordance with a quaternion command signal. This controller is based on optimal control (LQG) designed for a linearized model of the satellite dynamics and kinematics.

1) Control law implementation for B-dot: The B-dot controller has been implemented as PD controller for channel

(x,y,z) which is represented by the following transfer function:

$$H(s) = -80 \cdot \frac{12.6s + 2.69 \times 10^{-4}}{s + 12.6} \tag{12}$$

2) Control law implementation for Pointing: The pointing controller is a state-space controller which takes the angular velocity, ω, and an error quaternion, $\mathbf{q}_{1:3}$, as input and produces the commanded reaction wheel torque, N_{mv}, as output:

$$N_{mv} = K[\omega \ \mathbf{q}_{1:3}]^T \tag{13}$$

The gain matrix K has been found as an LQR controller based on a linearized model assuming the angular velocity is zero. This helps to ensure that the reaction wheels do not get saturated when canceling the angular momentum of the satellite body.

3) Transitions between Control Laws: The transition condition for going from B-dot to pointing is:

$$\sqrt{\omega^T \omega} < 0.017 \tag{14}$$

To go from the pointing controller and back again, a telecommand (i.e., an external event) is required.

B. Simulation Case

A detailed simulation model of the attitude control system for AAUSAT-II has been implemented in Simulink, based on [Amini *et al.*, 2005]. In the simulation case presented in this paper this Simulink model is used to simulate the satellite dynamics and kinematics, while the supervisory controller and associated control laws is implemented in Sophy, as depicted in figure 10. Furthermore, the *IOAdapter* interface is used to facilitate the exchange of data between Simulink and Sophy during the distributed simulation, as described in section V.

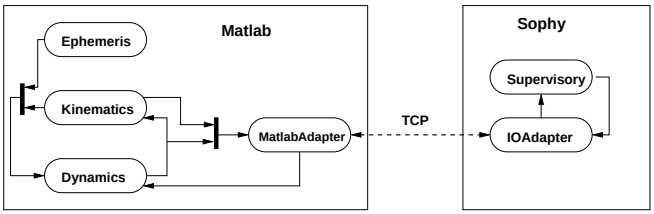

Fig. 11. Simulation setup distributed between Matlab and Sophy

For this simulation, the *IOAdapter* interface has been used to implement a plug-in for Sophy that exchanges data using a line oriented protocol running over a TCP socket connection. A library for Matlab has then been developed which through the use of S-functions communicates with this socket during the simulation. During the simulation Sophy is in charge of global time and dictates to what time Matlab should proceed in each step. The simulation has been carried out using a standard desktop PC, running both Simulink and Sophy.

C. Simulation Results

This subsection presents the results from the simulation case. Figure 12 shows the norm of the angular velocity. It can be seen that initially the B-dot controller slowly dissipates angular velocity and then at time 240s the threshold is reached and the pointing controller takes over. This results in a temporary rise in angular velocity as it controls the satellite to the proper attitude, where also the angular velocity reaches zero.

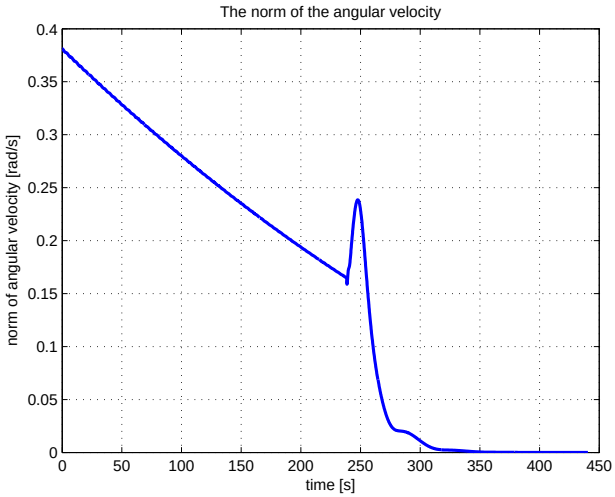

Fig. 12. The norm of the angular velocity

Figure 13 shows the corresponding attitude evolution. The reference for the controller is: $\mathbf{q} = [0 \ 0 \ 0 \ -1]$, which is quickly reached once the pointing controller takes over.

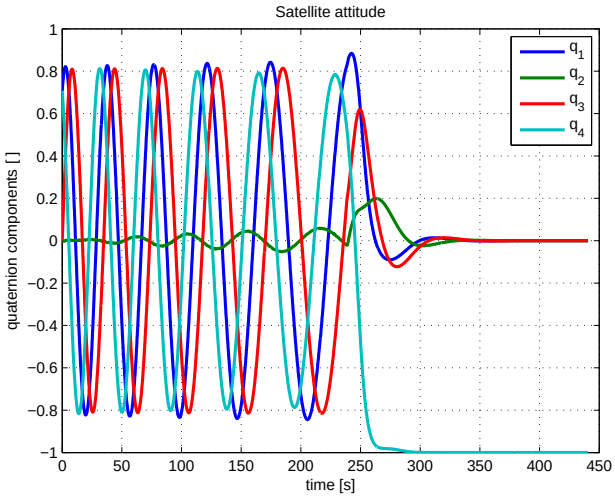

Fig. 13. The attitude Quaternion

Figure 14 shows the control input (voltages) to the magnetorquers. It can be seen that the controller is active until the pointing control takes over and that the amplitude of

the actuation gradually declines as angular velocity is slowly decreased.

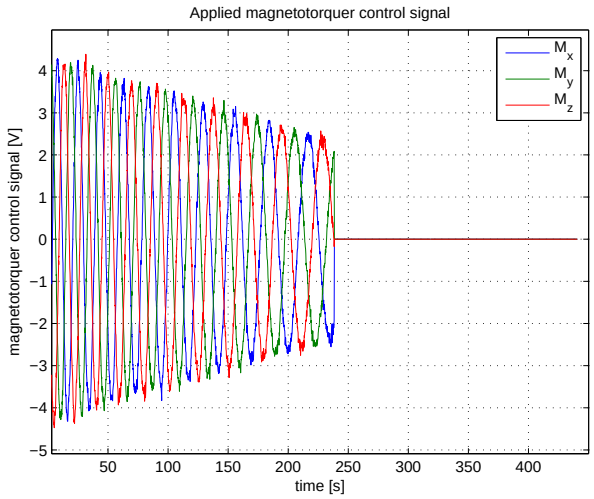

Fig. 14. Control signal to magnetometers

Figure 15 shows the command torque from the torque controller and it can seen how it becomes active at time 240s and quickly points the satellite in the correct direction.

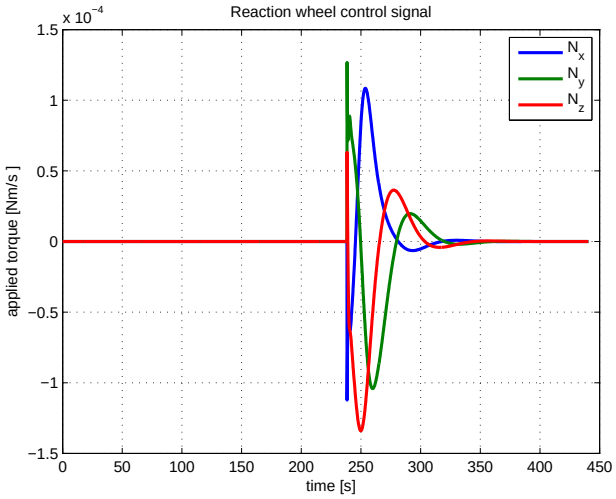

Fig. 15. Torque provided by reaction wheels

VII. Conclusions and Perspectives

This paper described the current status of the Sophy project with emphasis on its mathematical foundation in hybrid systems theory, along with its main interfaces for plug-in development, which open up possibilities for connectivity with many other systems. A case study focusing on supervisory control of a satellite attitude control system for a pico-satellite was presented, demonstrating Sophy's abilities to execute hybrid systems directly based on mathematical specifications and its ability to interface with external systems during run-time through the use of a generic plug-in architecture.

Future work will focus on deploying Sophy on other architectures more in line with the current computing capabilities on flight systems in order to analyze and optimize performance and make a thorough case study to pin-point any problems that must be addressed.

References

[Abildsten and Blanke, 1997] Soeren Abildsten and Mogens Blanke. *Fault Tolerant Control A Case Study of the Ørsted Satellite*. IEEE, 1997. Fault Diagnosis in Process Systems (Digest No: 1997/174), IEEE Colloquium on , 21 April 1997.

[Alminde *et al.*, 2004] Lars Alminde, Morten Bisgaard, Dennis Vinther, Tor Viscor, and Kasper Østergaard. *AAU-Cubesat Architectual Overview and Lessons Learned*. IFAC, 2004. Proceedings of the 16th IFAC Symposium on Automatic Control in Aerospace, 2004, Sct. Petersburg, Russia.

[Alminde *et al.*, 2006] Lars Alminde, Jan Dimon Bendtsen, and Jakob Stoustrup. *A Quantized State Approach to On-line Simulation for Spacecraft Autonomy*. American Institute of Aeronautics and Astronautics, 2006. In 2006 Modeling and Simulation Technologies Conference Proceedings. American Institute of Aeronautics and Astronautics, Keystone, Colorado, August 2006.

[Amini *et al.*, 2005] Rouzbeh Amini, Jesper A. Larsen, Roozbeh Izadi-Zamanabadi, and Dan D. V. Bhanderi. Design and implementation of a space environment simulation toolbox for small satellites. In *In Proc.: 25th International Astronautical Congress*, October 2005.

[Barton and Lee, 2002] Paul I. Barton and Cha Kun Lee. *Modeling, Simulation, Sensitivity Analysis, and Optimization of Hybrid Systems*. ACM, 2002. ACM Transactions on Modelling and Computer Simulation, Vol 12, No. 4, October 2002, pages 256-289.

[Branicky *et al.*, 1998] Michael S. Branicky, Vivek S. Borkar, and Sanjoy K. Mitter. *A Unified Framework for Hybrid Control: Model and Optimal Control Theory*. IEEE, 1998. IEEE Transactions on Automatic Control, Vol 43, NO. 1, January 1998.

[Consortium, 2006] World Wide Web Consortium. *XML Specification*. WWWC, 2006. http://www.w3.org/XML/.

[Grasso, 2002] C. A. Grasso. *The fully programmable spacecraft: procedural sequencing for JPL deep space missions using VML (Virtual Machine Language)*. IEEE, 2002. Aerospace Conference Proceedings Pages:1-75 - 1-81 vol.1.

[Henzinger, 1996] Thomas A. Henzinger. *The Theory of Hybrid Automata*. IEEE, 1996. Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on , 27-30 July 1996 Pages:278 - 292.

[Laursen *et al.*, 2005] Karl Kaas Laursen, Martin Fejrskov Pedersen, Jan Dimon Bendtsen, and Lars Alminde. *The SOPHY Framework: Simulation, Observation and Planning in Hybrid Systems*. IEEE, 2005. Fifth International Conference on Hybrid Intelligent Systems (HIS05), p 457-462.

[Lindholm and Yellin, 1999] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, 2nd Ed.* Sun Microsystems, 1999. http://java.sun.com/docs/books/vmspec/.

[Williams and Hofbaur, 2004] Brian C. Williams and Michael W. Hofbaur. *Hybrid Estimation of Complex Systems*. IEEE, 2004. IEEE Transactions on Systems, Man, and Cybernetics. Part B, Vol. 34 No. 5, October.