A Quantized State Approach to On-line Simulation for Spacecraft Autonomy

Lars Alminde*, Jan Dimon Bendtsen†, and Jakob Stoustrup‡

Future space applications will require an increased level of operational autonomy. This calls for declarative methods for spacecraft state estimation and control, so that the spacecraft engineer can focus on modeling the spacecraft rather than implementing all details of the on-line system. Celebrated model based methods such as Kalman filtering techniques and Model Predictive Control (MPC) rely on an on-line model of the system under control that can be simulated in faster than real-time. This becomes a severe challenge when the paradigm of modeling employed is that of hybrid systems where discrete and continuous dynamics co-exists.

This paper describes the design and implementation of an efficient engine for simulation of hybrid systems, specifically tailored for on-line applications. The simulation engine, contrary to traditional simulations systems, does not rely on discretization of time, but instead it works on a discretized state-space where the update of states is determined by a projection of points in time where the trajectory enters a new region. With this approach each state in the model is integrated separately, meaning that sparsity is exploited well. In addition hybrid transitions are located conservatively, i.e. without the need to ever "roll back" the simulation in time.

Nomenclature

Hybrid systems:

Q	location index set
X	continuous state-space
U	continuous input-space
Y	continuous output-space
E	input/output event labels
$\mathcal{F}$	forcing functions on the continuous state-space
$\mathcal{G}$	continuous output map
$\mathcal{T}$	transition map

DEVS Specification:

$\mathcal{X}$	inputs
$\mathcal{Y}$	outputs
$\mathcal{S}$	internal states
$\delta_{int}(\mathcal{S})$	state mapping function, for internal events
$\delta_{ext} : (e, \mathcal{S}, x)$	state mapping function for external events
$\lambda : \mathcal{S}$	output mapping
$ta : \mathcal{S}$	the time advance
e	duration since the last internal or external event
x	set of inputs

*PhD student, Section of Automation, Aalborg University, Fredrik Bajers Vej 7C, 9220 Aalborg, Denmark
†Associate professor, Section of Automation, Aalborg University, Fredrik Bajers Vej 7C, 9220 Aalborg, Denmark
‡Professor, Section of Automation, Aalborg University, Fredrik Bajers Vej 7C, 9220 Aalborg, Denmark

I. Introduction

In recent years strong attention has been put on so called *Hybrid Systems*, i.e. systems that are described by both continuous and discrete dynamics. It is the expressiveness of hybrid systems that makes them interesting for modeling highly complex systems. As a simple example consider the power management system of a satellite; The power generation from solar cells and the battery dynamics are well described using continuous Ordinary Differential Equations (ODE), however, the loads may switch in or out due to some logic dependent on the continuous state of the battery, hence discrete dynamics is coupled with the continuous dynamics.

Previous research into hybrid systems theory has focused on hybrid modeling, simulation and verification, see e.g.[1] More recent research has started to focus on the application of hybrid systems for control and estimation problems, e.g.[2–4] In terms of autonomy the potential of hybrid systems used with model based methods such as optimal nonlinear filtering and model predictive control is huge due to their expressiveness and generality.

This paper describes the design and implementation of an efficient engine for simulation of models of hybrid systems. The engine is specifically tailored for on-line applications. The simulation engine, contrary to traditional simulations systems, does not rely on discretisation of time, but rather it works on a discretized state-space where the update of states is determined by a projection of points in time where the trajectory enters a new region. This approach, called *Quantized State Systems* (QSS), was introduced in the paper by Kofmann.[5] With this approach each state in the model is integrated separately meaning that sparsity is exploited well, and it has been shown that this approach, for a given accuracy, is more efficient than traditional time-discrete methods.[6] The QSS approach can be implemented in the Discrete EVent Specification (DEVS), due to,[7] that specifies behavior of and interaction between atomic units communicating through message exchange.

The simulation engine is part of a multidisciplinary software framework under development at Aalborg University called SOPHY[8] (Simulation, Observation and Planning in HYbrid systems), where it implements key functionality in terms of hybrid simulation services for other components in the framework. The framework is developed as a research vessel for declarative control techniques for hybrid systems, with intended applications areas being spacecraft attitude control and formation flying.

The paper is structured as follows; At first definitions for hybrid systems are given, where after the Discrete EVent Specification (DEVS) is presented, which is used in the simulation architecture. Hereafter Quantised State Systems are presented as well as the mechanism for transition detection. This is followed by a discussion of SOPHY which integrates the work presented in the previous sections. Then an example is presented showing simulations of a simplified satellite power system. Finally the conclusions are given. The ideas as presented all implemented in the Java language as part of the SOPHY framework.

II. Hybrid Systems

This section presents the formal view of hybrid systems that is followed in the SOPHY project and continues to discuss the challenges associated with numerical integration of hybrid systems.

A. Definition of Hybrid Systems

Many different formal definitions of hybrid systems are available in the literature (see e.g.[1,3,4]). Within SOPHY there exist an abstract definition on which further restrictions wrt. expressivity can be imposed for specific classes of problems. This subsection first presents the abstract view and then continues to present the set of limitations imposed to define Hybrid Deterministic Systems (HDS), which will be the subject of this paper.

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

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

Where:
$Q = \{q \in \mathbb{Z}^+ | 1 \leq q \leq s\}$: is the set of location indexes with cardinal number $s \in \mathbb{Z}^+$
$X = \{\{x | x \in X_q\}_{q \in Q} | X_q = \mathbb{R}^{n_q}\}$: 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 $m_{q\in Q} \in \mathbb{Z}^+$
$Y = \{\{y|y \in Y_q\}_{q\in Q}|Y_q = \mathbb{R}^{o_q}\}$: is the continuous output-space with dimension $o_{q\in Q} \in \mathbb{Z}^+$
$E = \{e|e \in 2^\Sigma\}$: is the set of possible input/output event labels, where Σ is a set of labels

$\mathcal{F} : Q \times X \times U \mapsto \dot{X}$: is the forcing functions on the continuous state-space
$\mathcal{G} : Q \times X \times U \mapsto Y$: is a continuous output map
$\mathcal{T} : Q \times X \times U \times E \mapsto Q \times X \times E$: is a transition map

Remarks:

- Time is not explicitly given 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 time
- In most practical applications the dimensions of the state-, input-, and output-spaces will not change with different $q \in Q$
- The map $\mathcal{F}$, as defined above, allows Ordinary Differential Equations (ODE), but not e.g. differential algebraic or partial differential equations

1. Hybrid Deterministic System

The above definition is abstract and contains little information about how the maps are to be implemented in practice or how the initial state is defined. A Hybrid Deterministic System (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, further to make specification of the HDS more convenient the maps $\mathcal{F}$ and $\mathcal{G}$ will be defines as sets of functions with index $q \in Q$ and the transition map will be broken up into a set of different maps:

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

where:
Q, X, U, Y, E: are defined as before
$\mathcal{F} = \left\{\{f_q\}_{q\in Q} |\{q\} \times X_q \times U_q \mapsto \dot{X}_q\right\}$: 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
$\mathcal{T} = \left\{\{t_r\}_{r\in\{1,..,p\}} |Q \times X \times U \times E \mapsto Q \times X \times E\right\}$: are transition maps indexed from 1 to p

Where 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:
$j(\mathcal{S}_q) : \mathcal{S}_q \mapsto \{true, false\}$: is the *transition domain* which triggers the transition when true
$r(\mathcal{S}_q) : \mathcal{S}_q \mapsto Q \times X$: is an algebraic reset equation of the state
e_{in}: is an input event that causes the transition to trigger
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 logical combined inequalities, example:

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

B. Simulating Hybrid Systems

In each location the system must be simulated according to the ODE describing the dynamics until the state enters a transition domain. At this point a discrete jump is made to another location and the integration continues from there after reinitialization of the state-space. The challenges associated with this kind of integration of hybrid systems have been reported in[9] and.[3]

The major problem to solve when the hybrid trajectory evolves within one location is that of detecting when the transition functions, $j(\mathcal{S}_q)$, becomes true, i.e. signals that a discrete location jump must be executed.

1. State Event Detection

When integrating the ODE embedded at the current location, two types of events can occur; External events (also known as time events[3]) occur if e.g. an input variable changes value discretely. This can be accommodated by standard ODE solvers by making sure that a update is performed on the time of the event.

Another type of events are state events.[3] This type of event occurs when one of the transition equations becomes true. The solver will not know when this happens a priori. Therefore the solver must check for sign changes in all the transition equations after each update step to identify if a discrete location jump is to be made. However, this must be done consistently and the event must be located precisely in time. Consider Figure (1).

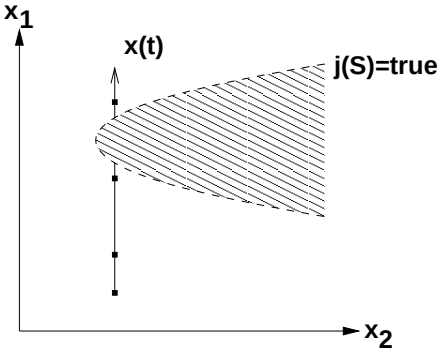

Figure 1. Trajectories of integration - missed transition detection

On the figure the dashed line indicates the transition domain and the full line is the state-space trajectory. The dots indicate where evaluations of $f_q(x, u)$ is made as part of the time discrete numerical integration. In this example a sign change of the transition equation will not be detected and the system evolves along a qualitatively wrong trajectory. Depending on the system type, this can prove quite fatal, especially if the system exhibits bifurcation or limit-cycle behavior.

It is therefore a fundamental requirement that a numerical integration algorithm for hybrid systems can provide a guarantee that transitions are always detected and taken.

2. Previous Work

Various extensions to standard ODE solvers have been considered to better handle the integration of hybrid systems. As stated above timed events can be handled by making sure that the integration performs an update at the exact time of the event.[3] State events have been accommodated by either choosing an artificial small step-size by the ODE solver or using so-called *discontinuity locking*.[3] The first approach leads to a very slow integration process not in line with the requirements of on-line simulation applications. In the latter approach the ODE is solved normally until a transition equation changes sign. At this point the standard integration stops and methods are employed to locate the transition within the last integration interval, e.g. using a bi-section algorithm[10] or finding roots of the transition equations.[11]

The latter approach is also called *roll-back*, because time will have to be rolled back to the time where the event occurs.[12] Compared to the method of artificial small step-sizes these methods are more efficient, but

they still can fail to detect events in scenarios like Figure (1) and there is still significant overhead whenever an event must be located.

In[12] the idea of looking at the step-size as a variable to be controlled via feedback was introduced as an extension to a linear multi step method for integrating ODEs and guarantees that the integration asymptotically approaches the transition without ever crossing it.

A method that guarantees not to cross over the transition domain during integration is called *conservative*, compared to the roll-back methods which are denoted as *positive*.[12]

In[13] extensive root-finding techniques were employed as an add-on mechanism to existing time discrete numerical integration algorithms with success, but at the expense of severely increasing the complexity of the integration algorithms.

III. DEVS Modeling

The Discrete Event Specification (DEVS) was formalized by Bernard P. Zeigler[14] as a language formalism for Discrete Event Systems (DES). Traditional DES descriptions, which enumerates all possible system configurations into a number of discrete states with associated transitions between. DEVS takes an alternative view and considers a number of units, called DEVS *atomic models*, that can implement very complex processing but interacts with other components through discrete interactions exchanging one or more values. DEVS has been applied for a wide range of modeling and simulations applications spanning from protocol verification to neural nets, see[15] for an overview.

More specifically, a DEVS model consists of a number of *atomic models* that accepts input events and generate output events. Each event is the communication of one or more real variables, which are associated to either an input port or an output port. Figure (2) depicts such a model with 3 input ports and 2 output ports.

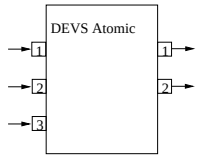

Figure 2. Atomic model with ports

These atomic models can be connected to form *coupled models*, which from the outside acts as an atomic model, i.e. DEVS models remain closed under coupling. The following will describe the specification of an atomic model and connections between these.

The following paragraphs described first in more detail atomic models and thereafter how models are coupled. The next section will describe its use for integration of hybrid systems.

A. Atomic DEVS Models

A general atomic DEVS model is specified as an 8-tuple:

$$\mathcal{M} = (\mathcal{X}, \mathcal{Y}, \mathcal{S}, \delta_{int}(\mathcal{S}), \delta_{ext}(e, \mathcal{S}, x), \lambda(e, \mathcal{S}), ta(\cdot)) \tag{5}$$

where:
$\mathcal{X}$: are the inputs. $x \in \mathcal{X}$ is a pair containing a port identifier and a value, i.e. $x = (port \in \mathcal{Z}^+, value \in \mathcal{R})$
$\mathcal{Y}$: are the inputs. $y \in \mathcal{Y}$ is a pair containing a port identifier and a value, i.e. $y = (port \in \mathcal{Z}^+, value \in \mathcal{R})$
$\mathcal{S}$: are the internal states[a]
$\delta_{int}(\mathcal{S}) : \mathcal{S} \rightarrow \mathcal{S}'$ is a state mapping function, for internal events
$\delta_{ext} : (e, \mathcal{S}, x) \rightarrow \mathcal{S}'$ is a state mapping function for external events
$\lambda : \mathcal{S} \rightarrow \mathcal{Y}$ is the output mapping
$ta : \mathcal{S} \rightarrow \mathbb{R}_+$ is the time advance function i.e. time to next internal event.
e: is the duration since the last internal or external event
x: is a set of inputs

The external event function, $\delta_{ext}()$, is called whenever the input to the model changes and the internal event function, i.e. an output event is generated by a connected model. $\delta_{int}()$ is called whenever the time advance function, $ta()$, becomes 0.

1. *Mapping Atomic Models to Software*

[a]it is up to the modeler to choose how $\mathcal{S}$ is represented

The specification of a DEVS *Atomic model* is captured in software as an abstract base class that is derived by all software objects that want to implement a DEVS *Atomic Model*. The most important attributes and methods of this class are shown in Figure (3). Whenever the class is inherited then the constructor in the new class defines initialization of the new class.

The *DevsMessage*, referenced in the class diagram, facilitates exchange of information between model components, i.e. it contains information paired in (port, value) objects.

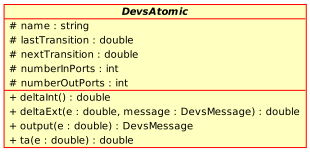

Figure 3. Most important features of the DevsAtomic class

B. Coupled Models

Atomic models can be coupled as specified in a coupling specification. Consider Figure (4), which shows two atomic models that are coupled. There are three model entities in the figure; $a1$ and $a2$ are internal atomic models and c is the coupled model.

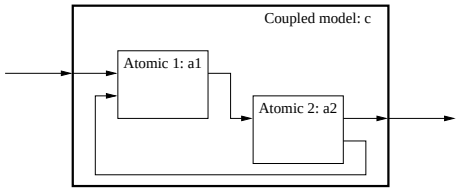

Figure 4. Coupled model

The coupling specification consists of three distinct sets with elements of the form $((m_1, p), (m_2, p))$, where $m_{1/2}$ specifies a model component and p the associated port number. The three sets are:[7]

EIC external input coupling, with $m_1 \in \{c\}$ and $m_2 \in \{a_1, ..., a_n\}$.

EOC external output coupling coupling, with $m_1 \in \{a_1, ..., a_n\}$ and $m_2 \in \{c\}$.

IC internal coupling, with $m_1 \in \{a_1, ..., a_n\}$ and $m_2 \in \{a_1, ..., a_n\}$.

Remark: Coupling is closed, hence a coupled model can itself be coupled with other atomic or coupled models.

1. *Mapping Coupled Models to Software*

The coupling specification is implemented in the *DevsCoordinator* class, which is shown in Figure (5) with its most important features. The class inherits behavior from *DevsAtomic* and extends its interface with functions for adding systems and connections to the coupled model. Since the class inherits the *DevsAtomic* class then itself can be added in other *DevsCoordinator* instantiations forming a hierarchy of coupled models.

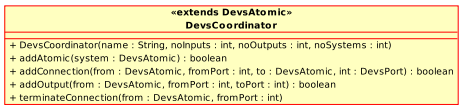

Figure 5. The DevsCoordinator class

In order for the coupled model to handle its contained *DevsAtomic* models it implements the following sketched procedure for advancing the states of the submodels, which forms the $\delta_{int}(\cdot)$ function call of the coordinator:

1. Form a set I of submodels with $ta(\cdot)$ equal to the least $ta(\cdot)$ of all submodels and advance time with this value
2. Call $\delta_{int}(\cdot)$ for all components in I
3. Call $\lambda(\cdot)$ for all components in I
4. from the set of generated outputs generate sets of input messages according to the coupling specification.
5. For each component with pending input call $\delta_{ext}(\cdot)$
6. Repeat from 1

IV. Quantized State Systems and Transition Handling

Typical numerical integration techniques are based on discretisation of time. Recently a consistent alternative approach has been developed which is based on discretisation of state values. This allows the integration process to be formulated as a discrete event process with the events specified at the times where a state crosses into another interval of the discretisation. The idea is sketched on Figure (6) where a continuous trajectory is shown together with the corresponding Quantized State Systems (QSS) trajectory. Note how the QSS trajectory is divided in time intervals according to the quantization of the value axis in contrast to traditional time discrete methods with constant time intervals. This means that computing power is being focused at states that exhibit fast dynamics.

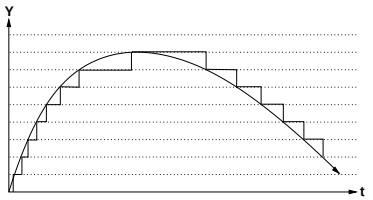

Figure 6. Idea of QSS integration

This method was originally developed by Bernard P. Zeigler and Herbert Praehofer[7] and recently extended by Ernesto Kofmann,[16] who formalized the QSS and the QSS2 methods, which are first and second order methods for numerical integration of ODEs, respectively. This section describe the general first order QSS method for integration of Ordinary Differential Equations (ODE) after.[16] Consider a time invariant ODE of the form[b]:

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{6}$$

The QSS method substitutes $\mathbf{x}(t)$ with a related function $\mathbf{q}(t)$:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{q}(t), \mathbf{u}(t)) \tag{7}$$

where $q_n = b(x_n))$ with $b(\cdot)$ being a hysteretic mapping $\mathbb{R} \rightarrow Q$. This hysteretic map imposes the following limitations on $q_n(t)$; Given a set of real numbers $Q = \{Q_i \in \mathbb{R} | Q_i < Q_j \ \forall i < j\}$ we define:[16]

$$q_n(t) = \begin{cases} Q_m \quad if \ t = t_0 \\ Q_{j+1} \, if \ x_n(t) = Q_{j+1} \wedge q_n(t^-) = q_j \\ Q_{j-1} \, if \ x_n(t) = Q_{j-1} \wedge q_n(t^-) = q_j \\ q(t^-) \, otherwise \end{cases} \tag{8}$$

with:

$$Q_m = Q_j | Q_j \leq x(t_0) < Q_{j+1} \tag{9}$$

This means that each continuous state in Equation (6) is approximated by a piecewise constant trajectory in Equation (7), which discretely changes its value exactly when the real state, x_n, enters a new interval in the set Q. The length of the interval: $\Delta Q = Q_{j-1} - Q_j$ is called the quantum and can be set for each state independently, such that it represents a meaningful change of the state.

A. ODE Simulation as a Discrete Event System

With the application of this hysteretic map then Equation (7) can be decoupled and simulated as indicated in Figure (7).

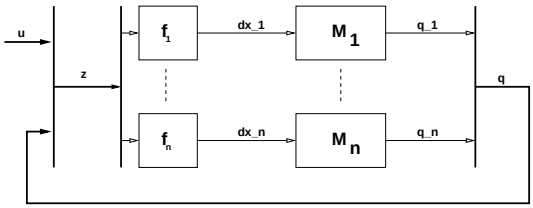

Figure 7. The structure for DEVS simulation

Here each function block, f_n, represents a row being calculated in $\mathbf{f}(\mathbf{q}(t), \mathbf{u}(t))$ and each model M_n represents an integrator that produces a piecewise constant output consistent with Equation 8. Each block

[b]If one wants to simulate a time depended ODE one can include an extra state representing time with no loss of generality

f_n is only called when there is an output event of a variable q_n on which the calculation of f_n depends. This helps to utilize sparsity in the system. No further details of the DEVS models of f_n will be presented here - it simply produces an new output when new inputs are received.

The following describes the operation of the models M_n in more detail; To simulate an ODE, δ_{ext} is called the corresponding state whenever there is a significant change in one of the input variables of the DEVS system (this includes both $\mathbf{q}(t)$ and $\mathbf{u}(t)$). δ_{int} will be called when there is a significant change in a state as predicted by the $ta(\cdot)$ function and finally $\lambda(\cdot)$ will evaluated whenever an output is required.

Each state S_n, where the index refers to the DEVS model for the n'th continuous state of the ODE, is considered a 4-tuple $S_n = (x_n, q_n, d_{x_n}, \sigma_n)$ with x_n being the predicted state value, q_n the last output of the integrator, $d_{x_n,}$ the derivative of x_n, and σ_n being the time to next event.

1. Internal Event Model, $\delta_{int}(\cdot)$

Internal events causes the following to happen for S_n:

$$x_n\prime \leftarrow q_n + sgn(d_{x_n})\Delta q_n \tag{10}$$

$$q_n\prime \leftarrow x_n\prime \tag{11}$$

$$d_{x_n}\prime \leftarrow d_{x_n} \tag{12}$$

$$\sigma_n\prime \leftarrow \begin{cases} \frac{\Delta q_n}{|d_{x_n}|} & if\ d_{x_n} \neq 0 \\ \infty & if\ d_{x_n} = 0 \end{cases} \tag{13}$$

Both x_n and q_n is set to the new quantized value and the time until the next event is calculated. The derivative d_{x_n} is left unchanged.

2. External Event Model, $\delta_{ext}(\cdot)$

The external event function is called whenever the input events are received On each event the following takes place with e being the time since last internal event for the corresponding state. The new input value (the derivative from f_n) is denoted u and is received as part of the event.

$$x_n\prime \leftarrow x_n + ed_{x_n} \tag{14}$$

$$d_{x_n}\prime \leftarrow u \tag{15}$$

$$q_n\prime \leftarrow q_n \tag{16}$$

$$\sigma_n\prime \leftarrow \begin{cases} \frac{\Delta q_n - (x_n\prime - q_n)}{u} & if\ u > 0 \\ \frac{\Delta q_n + (x_n\prime - q_n)}{|u|} & if\ u < 0 \\ \infty & if\ u = 0 \end{cases} \tag{17}$$

Equation (14 performs) forward Euler integration of the state. And Equation (17) calculates the remaining time in this quantized state.

3. Output Function, $\lambda(\cdot)$

Whenever the output function is called it generates:

$$\lambda_n(s_n, e) = q_n \tag{18}$$

4. Time Advance Function, $ta(\cdot)$

Whenever the time advance function is called it generates:

$$ta_n(e) = \sigma_n - e \tag{19}$$

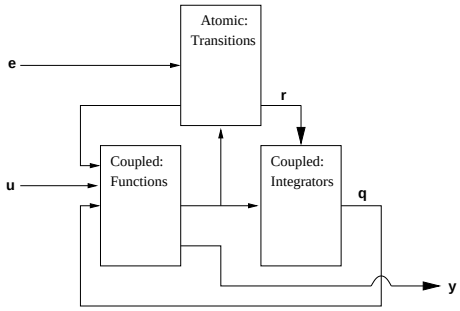

Figure 8. Transition handling in QSS/DEVS. Bold lines represent vector signals

B. Transition Handling in QSS/DEVS

In order to be able to integrate hybrid systems the QSS scheme for ODE integration is augmented with a mechanism to detect and enact transitions. This setting is depicted in Figure (8), where all the static functions f_n and M_n are grouped within one coupled DEVS model and similar for the integrators.

The new "Transitions" block internally integrates the state $\mathbf{x}$ from the input derivatives and predict when the transitions become enabled. At these points it generates an event connected to the static functions block, which causes it to load the dynamic equations for the new discrete location. At the same time the reset equations are calculated and the new state is supplied to the integrators, which adopt the new values.

Transitions ($r(\mathcal{S}_q)$ in Equation (3) can be caused by two mechanisms; The transition can be caused by an external event through the event channel e in which case the transition block promptly executes the reconfiguration event. Alternatively the transitions can be caused by an autonomous state event. The following paragraphs describes how the transition block calculates event times for autonomous transitions.

For this method we will require the transition equations to be linear. i.e. they can be written on the form (with $\mathbf{a} = [a_1 \ ... \ a_n]$ and b a scalar):

$$j(\mathbf{x}(t)) : b + \mathbf{a} \cdot \mathbf{x}(t) \geq 0 \tag{20}$$

Per se this seems rather restrictive, but a non-linear transition equation can in most cases be transformed to a linear equation by removing the non-linearity and introducing it as an extra state equation in the ODE of the current location.[12]

The transition block receives the state derivatives from the block of static functions. From these derivative the transition block integrates the current state $x(t)$. Now the transition equation is reformulated as:

$$j_*(\lambda) = j(\mathbf{x}(t)) + \lambda \cdot (\mathbf{a^T}\dot{\mathbf{x}}(t)) \tag{21}$$

where t is fixed at this time instant and $\dot{\mathbf{x}}$ is the vector of state derivatives received from the block representing static functions. By finding the root λ of this equation, it can be determined when the transition will become enabled.

$$\lambda = \frac{-j(\mathbf{x}(t))}{\mathbf{a^T}\dot{\mathbf{x}}(t)} \tag{22}$$

If λ is negative then the transition will never occur (as seen from this point in time). This procedure is repeated for all transitions τ relevant for the current location and the transition block sets its time advance function to the least positive value achieved for λ.

If there are no integrator transitions before this time then the transition will be executed next, otherwise the integration is performed where after the transition time calculations are iterated.

V. Simulation Architecture - SOPHY

The effort presented in this paper is part of an on-going effort called SOPHY for Simulation, Observation and Planning in Hybrid Systems, which is a research project with the goals to:

- Allow XML descriptions of systems and interconnections of systems

- Provide a framework for distributed processing at component level
- Provide various node services generated from received XML specifications, e.g.:
  - Simulation
  - Observation, Kalman filtering
  - Control, e.g. Model Predictive Control
- Allow research by basing core architectural components on exchangeable modules to play with different ideas

The architecture has been described in[8] and is implemented in Java. For a short review of the key concepts consider Figure (9). On this figure a simulation task is distributed across a network consisting of three computing nodes.

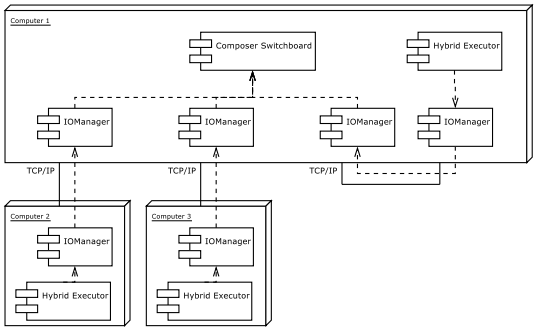

Figure 9. An example of SOPHY Deployment

Each node contains a *HybridExecutor* connected to the rest of the distributed simulation through an *IOManager*. The *IOManager* is a network abstraction and can be implemented for various network types, e.g. tcp/ip for desktop computers or CAN-bus for embedded systems.

One of the computing nodes contains the *ComposerSwitchBoard* which is responsible for distributing models and routing information between models simulated at different nodes. Upon initialization the *ComposerSwitchBoard* transmits simulation models to the nodes as XML files based on a user supplied XML description of the system which references the XML documents of the individual hybrid systems. The *HybridExecutor* receives the XML file and creates an appropriate simulation object from a number of possible plug-ins. The *ComposerSwitchBoard* then starts the simulation by advancing the global clock and distributing information flow between the nodes.

The remainder of this paper will focus on the DevsSophy plug-in that allows simulation of hybrid systems as described in the previous sections.

A. DevsSophy

The main packages can be seen on the diagram on Figure (10). The *DevsCore* package contains classes for DEVS simulation as described in Section (III), the *HybridCore* package contains all the functionality described Section in (IV) and finally *DevsTools* contains a class, *XMLModelFactory*, that translates XML descriptions of hybrid systems into coupled DEVS models by parsing the files and generating objects as required.

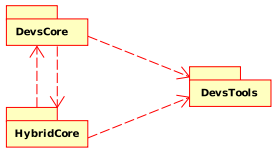

Figure 10. Package diagram for DevsSophy

When translated a structure such as depicted in Figure (11) is created. The black box contains the DEVS model parts that make up the model of the the hybrid system and the *DevsRunner* class is capable of executing the model as a stand alone model. Alternatively a class, *SophyRunner*, integrates the simulation with the SOPHY architecture as just described.

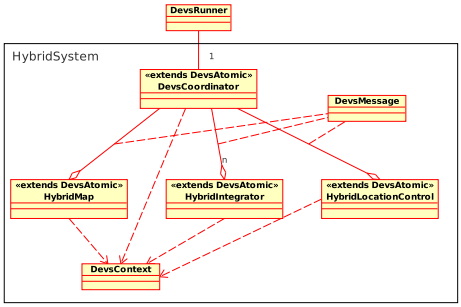

Figure 11. Class structure for a Hybrid System instantiated from an XML file

B. External Input to A Running Simulation

DevsSophy allows externally supplied inputs to be read from a comma separated file and injected into the inputs of the running simulation. This allows the use of data obtained from existing simulation environments to be incorporated and/or it allows the simulation to be run with different input sets.

The mechanism is based on a *DevsAtomic* instantiation which schedules its $\delta_{int}(\cdot)$ function at a constant sample-rate and each time reads in data from a specified file and produce it as output that then is distributed by the *DevsCoordinator*. This mechanism is used in the example in next section.

VI. Example Simulation - Simple Power System

As an example we will consider the simulation of a simplified power system for a satellite, see Figure (12). The simulation will take an external input, obtained from a Simulink simulation, representing the power input from solar panels from a slowly tumbling satellite in low Earth orbit.

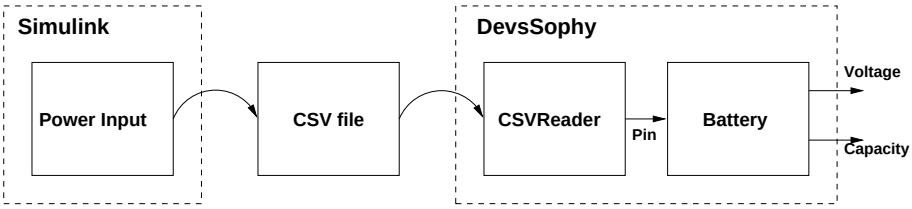

Figure 12. Simple example of a satellite power system

This power input is read into the DevsSophy environment as described in the last section by the *DevsAtomic* model *CSVReader* which communicates with the other *DevsAtomic* model which simulates the continuous dynamics of the battery as well as the load power consumption - this model is described as a hybrid model.

The battery model is depicted on Figure (13) and shows the three possible hybrid locations of the model as well as the transitions between them. C denotes the battery capacity state. A colon in the transition specification is followed by the reset specification ($r(\mathcal{S}_q)$ in Equation (3)). The battery in the simulation is of nominal capacity of $0.68Ah$ and nominal voltage of $24.8V$. The three locations are:

Nominal Here the satellite consumes $20W$ for platform systems. No payload is turned on and the battery charges whenever possible.

Payload When the battery is fully charged the satellite starts payload operations requiring $100W$ of power. The satellite does not charge in this location. The system returns to **nominal** at a specified capacity.

Safe Here the battery is close to depleted and only essential systems are on demanding $10W$. The battery charges whenever possible.

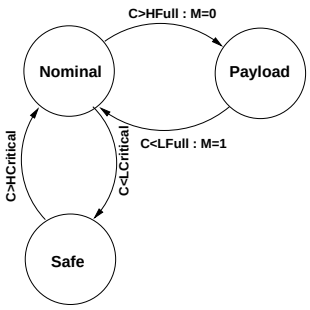

Figure 13. Discrete locations and transitions for the "battery" system

In all locations the continuous state is the battery capacity C and the output is the battery voltage V. These are specified as:

$$\dot{C}(t) = \frac{\frac{1}{3600}\left(M(t)P_{in} - P_{load}\right)}{V(t)} \tag{23}$$

$$V(t) = 18 + 10.6 \cdot C(t) \tag{24}$$

Here P_{load} is a constant in each location signifying the power consumption, as described above. $M(t)$ is a discrete state indicating if the system should attempt charging (in the **Nominal** or **Safe** location $M(t) = 1$, $M(t) = 0$ otherwise). This state is controlled by the resets associated with the transitions. The following XML snippet presents the specification of the hybrid system in the **Nominal** state:

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

       <diffequation state="M"> 0</diffequation>
       <diffequation state="C"> (0.00027*(Mode*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>
```

For a full specification of the *battery* hybrid system in the XML format see appendix A and compare to the definition of the Hybrid Deterministic System presented in Section (II).

A. Simulation Results

The results of the simulation of the simple power system is shown in Figure (14) for a simulation of $6000s$. The power input graph shows a varying input and a period of no input, which corresponds to eclipse. The voltage graph shows the variation of the voltage throughout the simulation.

Due to the affine relationship between the voltage and capacity in the model then the capacity graph is similar in form to the voltage graph. On the capacity graph the various transition limits are depicted from the hybrid model, e.g. at around $1200s$ it can be seen how the system for a short while enters the payload mode, before a depleted battery makes it return to nominal. The final graph shows the times where the hybrid location changes during the simulation and corresponds to crossings of the capacity state with transitions domains.

The simulation takes 1.8 seconds on a standard office computer and the quantum selected for simulation of the capacity is $5mAh$, i.e. Δq as described in Section (IV).

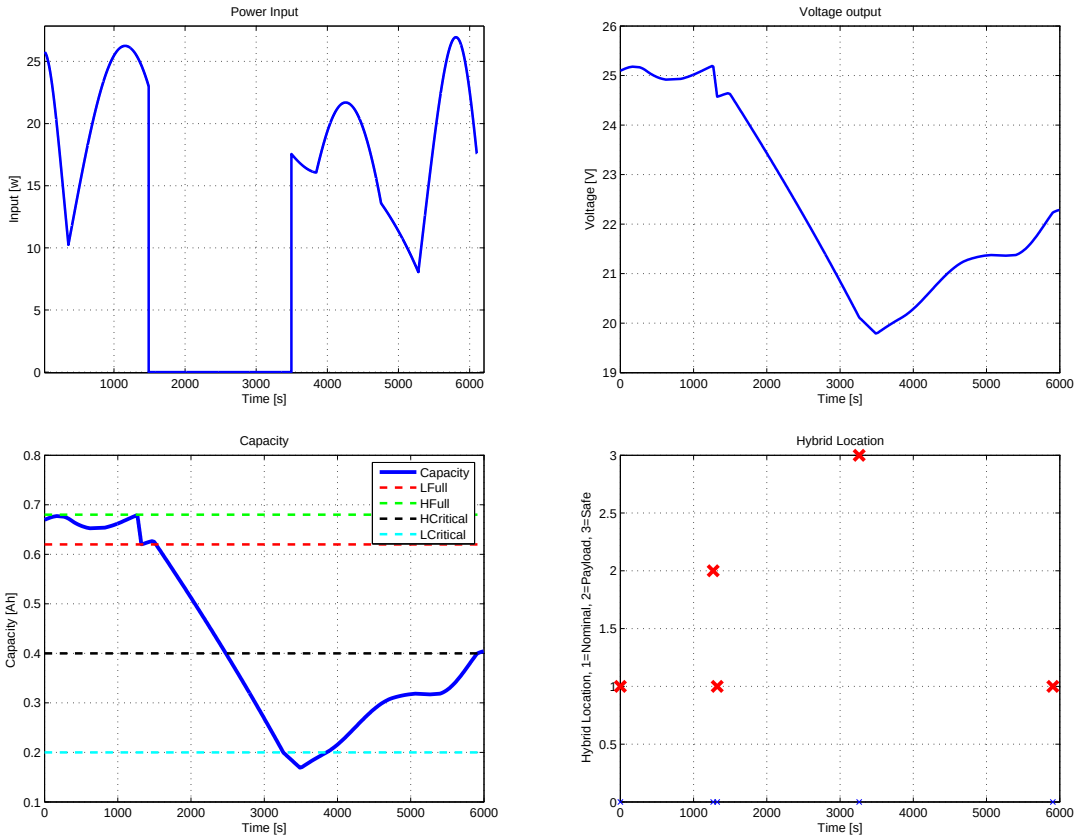

Figure 14. Results. UPPER-LEFT: Power input. UPPER-RIGHT: Battery Voltage. LOWER-LEFT: Battery Capacity. LOWER-RIGHT: Location changes

VII. Conclusion

This paper has presented an approach towards dealing with hybrid systems on-line, with the goal to make use of hybrid simulation as part of on-line control and estimation algorithms within the SOPHY framework.

To achieve consistent simulation of hybrid systems a number of ideas have been merged; A concise mathematical description of hybrid systems, the idea of quantized state systems, and the Discrete EVent Specification as a tool for discrete event simulation. The major contribution is the demonstration of automatic translation from the model domain to an on-line simulation.

Future work will extend the SOPHY architecture with means to communicate in real time with external processes, where after the architecture can be rolled out for control and estimation problems with the simulation capabilities at its heart.

Appendix A: Example XML Specification

The following shows an example of a SOPHY XML file, which specifies the "Battery&logic" block of the example. A number of documentation tags have been removed from the XML-code in order to conserve space.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SophySystem SYSTEM "SophySystem.dtd">
<SophySystem>
    <name>Battery</name>

    <constants>
```

```
        <constant symbol="HFull">0.68</constant>
        <constant symbol="LFull">0.62</constant>
        <constant symbol="HCritical">0.4</constant>
        <constant symbol="LCritical">0.2</constant>
    </constants>

<!-- ************************************************************** -->
    <states>
        <state>Capacity</state>
        <state>ChargeMode</state>
    </states>

    <inputs>
        <input>Pin </input>
    </inputs>

    <outputs>
        <output>V</output>
    </outputs>
    <locations>
<!-- ************************************************************ -->
        <location>
          <name>Nominal</name>

          <diffequation state="M"> 0</diffequation>
          <diffequation state="C"> (0.00027*(Mode*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>
<!-- ********************************************************** -->
        <location>
          <name>Payload</name>

          <diffequation state="M"> 0</diffequation>
          <diffequation state="C"> (0.00027*(Mode*Pin-100))/V</diffequation>
          <outputmap output="V"> 18+10.6*C </outputmap>

          <transitions>
```

```
            <transition>
              <name> toNominal</name>
              <domain>C &lt; LFull)</domain>
              <reset>
                <destination> Charge </destination>
                <statereset state="M">1</statereset>
              </reset>
            </transition>
          </transitions>
        </location>
<!-- ********************************************************** -->
        <location>
          <name>Safe</name>

          <diffequation state="M"> 0</diffequation>
          <diffequation state="C"> (0.00027*(Mode*Pin-20))/V</diffequation>
          <outputmap output="V"> 18+10.6*C </outputmap>
          <transitions>
            <transition>
              <name> toNominal</name>
              <domain>C &gt; HCritical)</domain>
              <reset>
                <destination> Nominal </destination>
              </reset>
            </transition>
          </transitions>
        </location>
    </locations>
</SophySystem>
```

References

[1]Henzinger, T. A., *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.

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

[3]Barton, P. I. and Lee, C. K., *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.

[4]Branicky, M. S., Borkar, V. S., and Mitter, S. K., *A Unified Framework for Hybrid Control: Model and Optimal Control Theory*, IEEE, 1998, IEEE Transactions on Automatic Control, Vol 43, NO. 1, January 1998.

[5]Kofman, E., *Discrete Event Simulation of Hybrid Systems*, SIAM, 2004, SIAM Journal on Scientific Computing. 25(5). pp 1771-1797.

[6]Zeigler, B. P., Jammalamadaka, R., and Akerkar, S., *Continuity and Change (Activity) Are Fundamentally Related In DEVS Simulation of Continous System*, Springer, 2005, Lecture notes in computer science, Volume 3397.

[7]Zeigler, B. P., Praehofer, H., and Kim, T. G., *Theory of Modelling and Simulation*, John Wiley and Sons, 2000, 2nd edition.

[8]Laursen, K. K., Pedersen, M. F., Bendtsen, J. D., and Alminde, L., *The SOPHY Framework: Simulation, Observation and Planning in Hybrid Systems*, IEEE, 2005, Fifth International Conference on Hybrid Intelligent Systems (HIS05), p 457-462.

[9]Mosterman, P. J., *An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages*, Springer, 1999, Hybrid Systems: Computation and Control '99, Lecture Notes in Computer Science vol. 1569.

[10]Taylor, J. H. and Kebede, D., *A Rigorous Hybrid Systems Simulation of an Electro-mechanical Pointing System with Discrete-time Control*, AAAC, 1997, Proceedings of the American Control Conference, Alberquerque, June 1997, page: 2786-2789.

[11]Lieu, J., Liu, X., Koo, T.-K. J., Sinopoli, B., Sastry, S., and Lee, E. A., *A Hierarchical Hybrid System Model and Its Simulation*, IEEE, 1999, Proceedings of the 38th Conference on Decision and Control, Phoenix, December 1999, page 3508-3513.

[12]Esposito, J. M. and Kumar, V., *An Asynchronous Integration and Event Detection Algorithm for Simulating Multi-Aganet Hybrid Systems*, ACM, 2004, ACM Transactions on Modelling and Computer Simulation, Vol 14, No. 4, October 2004, pages 363-388.

[13]Shampine, L. F., Gladwell, I., and Brankin, R. W., *Reliable Solution of Special Event Location Problems for ODEs*, ACM, 1991, ACM Transactions on Mathematical Software, Vol 17 No 1, March 1991, pp. 11-25.

[14]Zeigler, B. P., *Theory of Modelling and Simulation*, John Wiley and Sons, 1976, 1st edition.

[15]Sarjoughian, H. S. and Cellier, F. E., *Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies*, Springer Verlag, 2001.

[16]Kofman, E., *Discrete Event Simulation of Hybrid Systems*, Society for Industrial and Applied Mathematics, 2004, SIAM Journal on Scientific Computing Volume 25, Number 5, pp. 1771-1797.