MODELING AND SIMULATION OF DIGITAL CONTROLLERS FOR HYBRID DYNAMIC STRUCTURE SYSTEMS

Fernando J. Barros
Dept. Eng. Informática
Universidade de Coimbra
3030 Coimbra, Portugal

ABSTRACT

Complex systems exhibiting structural changes can be better represented by models that can mimic these transformations. The *Heterogeneous Flow System Specification* (HFSS) is a comprehensive formalism that can represent a large variety of models using a unifying representation. The HFSS formalism represents models in a hierarchical and modular form. The explicit representation of structure makes possible to alter it dynamically. Among hybrid models, the most important include digital controllers and hybrid integrators. The HFSS ability to provide a common ground to represent all these elements permits to model complex systems in a simple framework. We present a detailed model of a PID controller and we show how this digital controller can be merged with other types of components. To illustrate formalism application we use a dynamic structure network of hybrid components to model a 2-stage rocket system, whose velocity is set by a PI digital controller. Simulation results for the rocket system are presented.

1. INTRODUCTION

The representation of hybrid dynamic structure systems is a challenging problem to modeling and simulation methodologies. To help modeling these kind of systems we have developed the *Heterogeneous Flow System Specification* (HFSS) modeling and simulation formalism a novel and comprehensive framework for representing hierarchical and modular hybrid systems with adaptive structure. Examples include switched systems and mobile agent systems.

To show the ability of the HFSS formalism to represent hybrid systems we model a two-stage rocket whose velocity is set by a digital controller. This model exhibits structural changes corresponding to rocket split when the first stage runs out of fuel. The ability to mimic system structure leads to better to understand rocket model. The model exhibits both continuous and discrete behavior. While rocket position and velocity are continuous variables regulated by differential equations, the digital controller introduces discontinuities in the rocket thrust that are represented by discrete event signals. The structural changes occurring in the rocket are also represented by discrete signals that originate model variation.

Contrarily to discrete event systems, that have an exact and simple representation in digital computers, continuous systems can only be represented with some error. Discrete time machines are the most widely used formalism to represent numerical methods in digital computers. However, this formalism can only deal with a single time step, being unusable to provide the integration of components requiring independent time steps. This situation occurs, for example, when digital controllers need to be merged with ODEs solvers. This problem becomes even harder to solve when there is a need to use asynchronous solvers and/or digital controllers with variable sampling rates. Currently, most common tools are based on pure modeling formalisms translating models into *ad hoc* code specific of the tool vendor.

To permit asynchronous time steps we have developed the concept of continuous flow and we have created the Continuous Flow System Specification (CFSS). CFSS formalism supersedes the traditional discrete time formalism allowing the representation of continuous systems with independent and variable time steps. CFSS models have a precise semantic enabling an algorithmic description of their simulators. This characteristic makes CFSS a modeling and simulation formalism enabling model interoperability, essential for supporting model communication in an infrastructure like the High Level Architecture (HLA) [5].

The HFSS formalism uses CFSS for representing continuous systems and it is able to integrate continuous and discrete types of models by offering a unifying representation to all components. The usual representation of hybrid systems assumes they are a combination of differential equations and discrete event systems [7], [8]. Thus, sample based systems like digital controllers have not been considered. Actually, these formalisms are intended for modeling and analysis and the inclusion of digital controllers would preclude a formal analysis. On the contrary, the HFSS formalism was designed for simulation purposes and it is intended to represent a larger number of systems.

A particular structural change that has attracted the research on hybrid systems is commonly referred by switching systems [6]. These systems are controlled by a set of differential equations corresponding to a particular model of operation. When these conditions change, a

new mode, corresponding to a new set of equations, is chosen. Many formalisms are limited to a finite set of modes and thus they cannot represent complex systems where there is an exponential explosion of these modes. Examples of such systems include electrical circuits with a large number of switching devices like diodes or thiristors. In these cases, the incremental changes enabled by the HFSS formalism can offer a more suitable representation. In general, formalisms tailored to represent structural changes based on an explicit representation of all possible structures cannot deal with an unbounded number of different model configurations [1].

2. HETEROGENEOUS FLOW COMPONENTS

The Heterogeneous Flow System Specification (HFSS) is a formalism aimed to represent piecewise constant partial state components. The HFSS formalism combines the CFSS formalism [2], to represent continuous flow systems and the DEVS formalism [9] to represent discrete flow systems. HFSS systems can accept both continuous and discrete input flows and produce both continuous and discrete output flows [3]. The HFSS formalism has been shown to be closed under the coupling operation enabling hierarchical model definition [3]. The abstract simulators define a non-ambiguous network structure during the whole simulation by imposing a precise semantics for handling simultaneous events [4].

2.1. Basic Model

A basic model in the *Heterogeneous Flow System Specification* (HFSS) is formally defined by

$$HFSS = (X, Y, S, \rho, \tau, q_0, \delta, \Lambda_c, \lambda)$$

where

$X = X_c \times X_d$ is the set of input values with
- X_c is the set of continuous input values
- X_d is the set of discrete input values

$Y = Y_c \times Y_d$ is the set of output values with
- Y_c is the set of continuous output values
- Y_d is the set of discrete output values

S is the set of partial states (p-states)

$\rho: S \to \mathbf{R}_0^+$ is the time-to-input function

$\tau: S \to \mathbf{R}_0^+$ is the time-to-output function

$Q = \{(s,e) | s \in S, 0 \le e \le \nu(s)\}$ is the state set

$\nu: S \to \mathbf{R}_0^+$ is the time-to-transition function defined by

$$\nu(s) = \min\{\rho(s), \tau(s)\}$$

$q_0 = (s_0, e_0) \in Q$, is the initial state

$\delta: Q \times (X_c \times X_d^\phi) \to S$ is the transition function, with

$X_d^\phi = X_d \cup \{\phi\}$

$\Lambda_c: Q \to Y_c$ is the continuous output function

$\lambda: S \to Y_d^\phi$ is the partial discrete output function

The discrete output function, $\Lambda_d: Q \to Y_d^\phi$, is defined by

$$\Lambda_d(s,e) = \begin{cases} \lambda(s) & \text{if } e = \tau(s) \\ \phi & \text{if } e < \tau(s) \end{cases}$$

The output function, $\Lambda: Q \to Y_c \times Y_d^\phi$ is defined by

$$\Lambda(q) = (\Lambda_c(q), \Lambda_d(q))$$

A HFSS components have piecewise constant partial states making them realizable in digital computers. The component state is defined the pair (s,e), where e is the time elapsed in the current partial state s. Continuous flows are sampled at times defined by function ρ. Discrete flow values are produced at times defined by function τ. Changes in a component state occur whenever a sample or a discrete flow is produced. State changes can also be triggered by the arrival of discrete flow at component input. Component new state is described by the transition function δ. A more detailed description of the semantics of HFSS basic models is given in [3].

2.2. Network Model

The HFSS formalism can represent models with a time-varying structure. These dynamic structure models offer a more intuitive representation of real systems for they are able to mimic the dynamic creation and destruction of entities, and the dynamic nature of the relationship among entities within a system. Formally, a Heterogeneous Flow System Specification Network is a 6-tuple

$$HFN_N = (X_N, Y_N, \eta, M_\eta)$$

where

N is the network name

$X = X_{c_N} \times X_{d_N}$ is the set of input values, with
- X_{c_N} the set of continuous input flow values
- X_{d_N} the set of discrete input flow values

$Y = Y_{c_N} \times Y_{d_N}$ is the set of output values, with
- Y_{c_N} the set of continuous output flow values
- Y_{d_N} the set of discrete output flow values

η is the name of the dynamic structure network executive

M_η is the model of the executive η

The model of the executive is a modified HFSS, defined by

$$M_\eta = (X_\eta, Y_\eta, S_\eta, \rho_\eta, \tau_\eta, q_{0,\eta}, \gamma, \Sigma^*, \delta_\eta, \Lambda_{c\eta}, \lambda_\eta)$$

where

Σ^* is the set of network structures

$\gamma: Q_\eta \to \Sigma^*$ is the structure function

The network structure $\Sigma_{j,e} \in \Sigma^*$, corresponding to the state $(s_{j,\eta}, e) \in Q_\eta$, is given by the 4-tuple

$$\Sigma_{j,e} = \gamma(s_{j,\eta}, e) = (D_j, \{M_{i,j,e}\}, \{I_{i,j}\}, \{Z_{i,j,e}\})$$

where

D_j is the set of component names associated with the executive state $(s_{j,\eta}, e)$

for all $i \in D_j$
- $M_{i,j,e}$ is the model of component i
- $I_{i,j}$ is set of components influencers of i
- $Z_{i,j,e}$ is the input function of component i

For simplicity we assume here that the models do not change with executive elapsed time e and thus, $M_{i,j,e} = M_{i,j}$.

These variables are subject to the following constraints for every $q_{j,\eta} \in Q_\eta$:

$\eta \notin D_j, N \notin D_j, N \notin I_{N,j}$

$M_{i,j} = (X_{i,j}, Y_{i,j}, S_i, \rho_i, \tau_i, q_{0,i}, \delta_{i,j}, \Lambda_{c\,i,j}, \lambda_{i,j})$ is a basic HFSS model, for all $i \in D_j$, with

$$\delta_{i,j}: Q_i \times X_{i,j} \to S_i$$

$$Z_{i,j}: \bigtimes_{k \in I_{ij}} V_{k,j} \to X_{i,j}, \text{ for all } i \in D_j \cup \{\eta\}$$

where

$$V_{k,j} = \begin{cases} Y_{k,j} & \text{if } k \neq N \\ X_N & \text{if } j = N \end{cases}$$

The network output function is given by

$$Z_{N,j}: \bigtimes_{k \in I_{N,j}} Y_{k,j} \to Y_N$$

Changes in network structure include the ability to change network composition and coupling and they are achieved by changing executive state. The mapping from executive state into network structure is made by function γ. Network semantics enable incremental operators to dynamically adapt the structure [3]. Components remaining in the network keep their state unchanged. On the contrary, new components are initialized using their initial state q_0. This behavior permits to easily define *add* and *remove* operators to modify network composition. Structural changes can be made by expressing the modifications without the need to define completely the new network structure. This feature permits to define networks with a possibly infinite number of different configurations. A more detailed description of the semantics of HFSS networks is given in [3].

3. PID DIGITAL CONTROLLER

We describe the model of a PID digital controller that is used in this paper to control the velocity of the 2-stage rocket presented in the next chapter. The digital controller considered here samples its continuous input at a fixed rate and produces both continuous and discrete event flows. A zero-holder PID digital controller with parameters P = 40.0, I = 1.0, D = 1.0, and a fixed sampling period (sp) of 1.0 s can be represented by the HFFS model given by:

$$PID = (X, Y, S, \rho, \tau, q_0, \delta, \Lambda_c, \lambda)$$

where

$X = \mathbb{R} \times \{\}$

$Y = \mathbb{R} \times \mathbb{R}$

$S = \{\text{run, out}\} \times \mathbb{R}^{10}$

$\rho(phase, \alpha, \beta, P, I, int, D, der, xc1, xp1, sp) = \alpha$

$\tau(phase, \alpha, \beta, P, I, int, D, der, xc1, xp1, sp) = \beta$

$q_0 = ((\text{run}, 0.0, \infty, 40.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0), 0.0)$

$\delta((\text{run}, \alpha, \beta, P, I, int, D, der, xc1, xp1, sp), e), (xc, \phi)) =$

$(\text{out}, \infty, 0, P, I, int + e \times (xc + xc1) / 2\ , D, (xp - xp1) / e, xc, xp, sp)$

$\delta((\text{out}, \alpha, \beta, P, I, int, D, der, xc1, xp1, sp), e), (xc, \phi)) =$

$(\text{run}, sp, \infty, P, I, int, D, der, xc1, xp1, sp)$

$\Lambda_c((phase, \alpha, \beta, P, I, int, D, der, xc1, xp1, sp), e) =$

$P \times xc1 + I \times int + D \times der$

$\lambda(phase, \alpha, \beta, P, I, int, D, der, xc1, xp1, sp) =$

$P \times xc1 + I \times int + D \times der$

The sampling period of the controller is defined by value sp. A change in this parameter would transform this controller into a variable sampling rate one. In this case the controller samples its input at a fixed rate sp and it produces both continuous and discrete flow outputs. For simplicity we consider here the controller continuous output flow as piecewise constant. Using the variable e, present at the definition of function Λ_c would allow the description of a variable output flow value, like a first–order value for example.

Other models like variable step numerical integrators can also be described in the HFSS formalism and a description of a first-order integrator can be found in [4].

4. A 2-STAGE ROCKET

We describe now a 2-stage rocket whose velocity is set by a PI digital controller. The rocket has one fuel tank in each stage. When the first stage is empty, it is separated from the rocket head and it starts falling obeying to the laws of gravity and friction. At this point, the 2nd stage continues climbing with the help of its own fuel. When the fuel in the 2nd stage finishes, the rocket head starts falling obeying to the laws of gravity and friction.

Rocket velocity is set by the PI controller that works to keep velocity constant. The controller has two different set of parameters for the two rocket configurations. The reference velocity is 150 m/s when both stages are present. This value is reduced to 100 m/s when only the 2nd stage is present.

Rocket representation is made in Figure 1 where the entire rocket and the separated stages are depicted. The rocket is described by the following parameters:

Drag of the first stage (tank): 0.01 Kg/m
Mass of the first stage (empty): 20 Kg
Fuel of the first stage: 130 Kg
Drag of the second stage (rocket head): 0.005 Kg/m
Mass of the second stage (empty): 10 Kg
Fuel of the second stage: 60 Kg
Ratio: mass consummation / force: 0.0025 Kg/s/N

The 0th–holder PI controller tries to set the rocket speed at 150 m/s with parameters P = 60 and I = 4. Rocket

maximum thrust is 6000 N. Sampling rate is fixed and set to 1.0 s. Controller parameters are changed when the first stage is dropped and the new values are P = 30 and I = 5. The new reference velocity is 100 m/s.

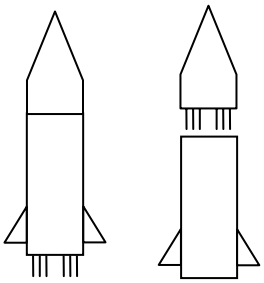

Figure 1. 2-Stage rocket.

To represent the described model we use the SMALLCAOS modelling and simulation environment, a Smalltalk implementation of the HFSS formalism. There classes are used in this model. Class **AB** represents integrators based on a variant of the Adams-Bashforth 3rd order method with an adaptive step-size. Class **PID** implements a modified PID digital controller described in Section 3 and enhanced to receive new control parameters at run-time. Class **Detector** is a zero cross detector. Initial model configuration is given below:

```
start
"-------------------------------------------------------------------"
    super start.
    phase := #zero.
    f1 := 130.0.
    m1 := 20.0.
    f2 := 60.0.
    m2 := 10.0.
    dragH := -0.005.          "Head drag"
    dragT := -0.01.           "Tank drag"
    self simTime: 210.0 nEvents: 12000.
"-------------------------------------------------------------------"
    self add: AB name: #Y1 initialState: [:c|
        c ode: [:x :y|x] y0: 0.0.
    ].
    self add: AB name: #VY1 initialState: [:c|
        c ode: [:x :y|x] y0: 0.0.
    ].
"-------------------------------------------------------------------"
    self add: PID name: #T initialState: [:pid|
        pid samplingRate: 1.0.
        pid P: 60.0 I: 4.0 D: 0.0 min: 0.0 max: 6.0e3
    ].
"-------------------------------------------------------------------"
    self model: #T addIFC: #VY1.
    self model: #T inputFunctionC: [:v :_|150.0 - v].
"-------------------------------------------------------------------"
    self model: #T addIFD: #Executive.
    self model: #T inputFunctionD: [:x :_|x].
"-------------------------------------------------------------------"
    self add: AB name: #F1 initialState: [:c|
        c ode: [:x :y|x] y0: f1.
    ].
    self model: #F1 addIFC: #T.
    self model: #F1 inputFunctionC: [:th :_| th * -0.0025].
    self model: #F1 addIFD: #T.
    self model: #F1 inputFunctionD: [:th :_|th].
"-------------------------------------------------------------------"
    self add: Detector name: #D.
    self model: #D addIFC: #F1.
    self model: #D inputFunctionC: [:f :_|f].
    self model: #D addIFD: #T.
    self model: #D inputFunctionD: [:th :_|th].
"-------------------------------------------------------------------"
    self model: #Y1 addIFC: #VY1.
    self model: #Y1 inputFunctionC: [:vy :_|vy].
"-------------------------------------------------------------------"
    self model: #Y1 addIFD: #T.
    self model: #Y1 addIFD: #D.
    self model: #Y1 inputFunctionD: [:th :v :_|true].
"-------------------------------------------------------------------"
    self model: #VY1 addIFC: #VY1.
    self model: #VY1 addIFC: #F1.
    self model: #VY1 addIFC: #T.
    self model: #VY1 inputFunctionC: [:vy :f :th :_| |mass|
        mass := m1 + m2 + f2 + f.
        (dragH * vy * (vy abs) + th) / mass - 9.8
    ].
"-------------------------------------------------------------------"
    self model: #VY1 addIFD: #D.
    self model: #VY1 inputFunctionD: [:v :_|v].
"-------------------------------------------------------------------"
    self model: #Executive addIFC: #Y1.
    self model: #Executive addIFC: #VY1.
    self model: #Executive inputFunctionC: [:y1 :vy1 :_|
        Array with: y1 with: vy1
    ].
"-------------------------------------------------------------------"
    self model: #Executive addIFD: #D.
    self model: #Executive inputFunctionD: [:d :_|d].
"-------------------------------------------------------------------"
```

Rocket position id represented by component #Y1 and rocket velocity by #VY1. Fuel tank is represented by #F1 and the digital controller is represented by component #T. The fuel detector is represented by component #D1 that signals when the fuel mass becomes zero. To add a component to the network one needs to specify component class and component name. To add the Adams-Bashforth integrator named #Y1 the following call is made:

```
self add: AB name: #Y1 initialState: [:c|
    c ode: [:x :y|x] y0: 0.0.
].
```

where AB is the integrator class, the parameter "ode" represents the differential equation and "y0" the initial value of rocket position.

Coupling between the modular components is set by methods addIF{C|D} and inputFunction{C|D}. The former method sets the influencers of a component. The latter sets component input function that translates influencers outputs into component input. SMALLCAOS has two pairs of methods with prefix C and D to differentiate between continuous and discrete flows, respectively. For example, the commands:

```
self model: #F1 addIFC: #T.
self model: #F1 inputFunctionC: [:th :_| th * -0.0025].
```

indicate that fuel tank #F1 is influenciated by the PID #T and its value is decreased by th * 0.0025 where th is the thrust value set by the controller. The block diagram of rocket initial structure is given by Figure 2, where each circle corresponds to a component input function. The first structural change occurs when the first stage runs out of fuel. This structural change is described by the following SMALLCAOS excerpt of the executive transition function:

```
delta: e_ xc: xc xd: xd
    (xd isNil) ifTrue: [^self passivate].
    (phase = #zero) ifTrue: [
        phase := #one.
        "---------------------------------------"
        self model: #VY1 removeIFC: #F1.
        self model: #D removeIFC: #F1.
        self remove: #F1.
        out := Array with: 1.0 with: 30.0 with: 5.0 with: 0.0.
        beta := 0.0.
        self model: #T inputFunctionC: [:v :_|100.0 - v].
        "---------------------------------------"
        self add: AB name: #F2 initialState: [:c|
            c ode: [:x :y|x] y0: f2.
        ].
        self model: #F2 addIFC: #T.
        self model: #F2 inputFunctionC: [:th :_| th * -0.0025].
        self model: #F2 addIFD: #T.
        self model: #F2 inputFunctionD: [:th :_|th].
        self model: #VY1 addIFC: #F2.
        self model: #D addIFC: #F2.
        "---------------------------------------"
        self model: #VY1 inputFunctionC: [:vy :th :f :_| |mass|
            mass := m2 + f.
            ((dragH * vy * (vy abs) + th)/ mass) - 9.8
        ].
        "---------------------------------------"
        self add: AB name: #Y2 initialState: [:c|
            c ode: [:x :y|x] y0: (xc at: 1).
        ].
        self add: AB name: #VY2 initialState: [:c|
            c ode: [:x :y|x] y0: (xc at: 2).
        ].
        "---------------------------------------"
        self model: #Y2 addIFC: #VY2.
        self model: #Y2 inputFunctionC: [:vy :_|vy].
        self model: #VY2 addIFC: #VY2.
        self model: #VY2 inputFunctionC: [:vy :_|
            (dragT * vy * (vy abs) / m1) - 9.8
        ].
        "---------------------------------------"
        self add: Detector name: #D2.
        self model: #D2 addIFC: #Y2.
        self model: #D2 inputFunctionC: [:y2 :_|y2].
        "---------------------------------------"
        self model: #Executive addIFD: #D2.
        self model: #Executive inputFunctionD: [:d1 :d2 :_|
            Array with: d1 with: d2
        ].
        "---------------------------------------"
        self add: Detector name: #D.
        self model: #D addIFC: #F1.
        self model: #D inputFunctionC: [:f :_|f]."
        "---------------------------------------"
        ^nil
    ].
```

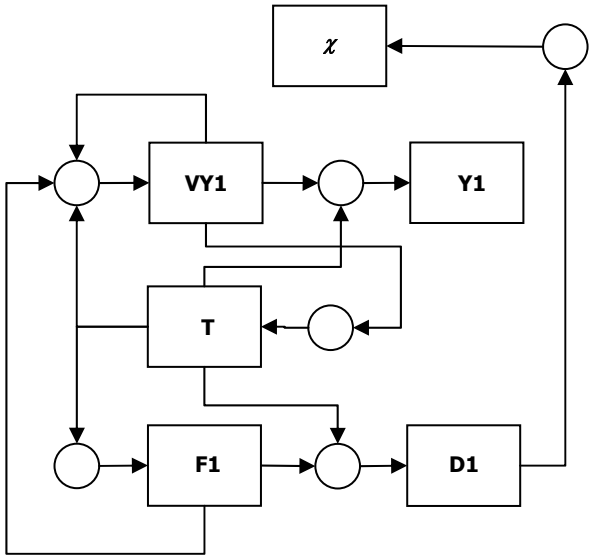

Figure 2. Rocket initial configuration.

The structural changes involve removing the empty tank #F1, adding the fuel tank #F2. Removal of component #F1 is made by the call:

```
self remove: #F1.
```

The removed tank is replaced by new components #VY2 and #Y2 that represent the falling tank velocity and position, respectively. This falling tank is modelled as a free fall body obeying to rules of gravity and friction. We consider friction proportional to the square of the velocity. The PID controller named #T receives at this time a new set of parameters (sampling rate = 1.0, P = 20.0, I = 1.0). The reference velocity is set to the new value 100.0 m/s by the method call:

```
self model: #T inputFunctionC: [:v :_|100.0 - v].
```

Detector #D1 is now connected to component #F2 that represents 2nd stage fuel. This detector will signal when the rocket head runs out of fuel. The block diagram of the split rocket is given by Figure 3. Component #D2 is added to detect the instant the 1st stage will reach the ground (Y2 = 0).

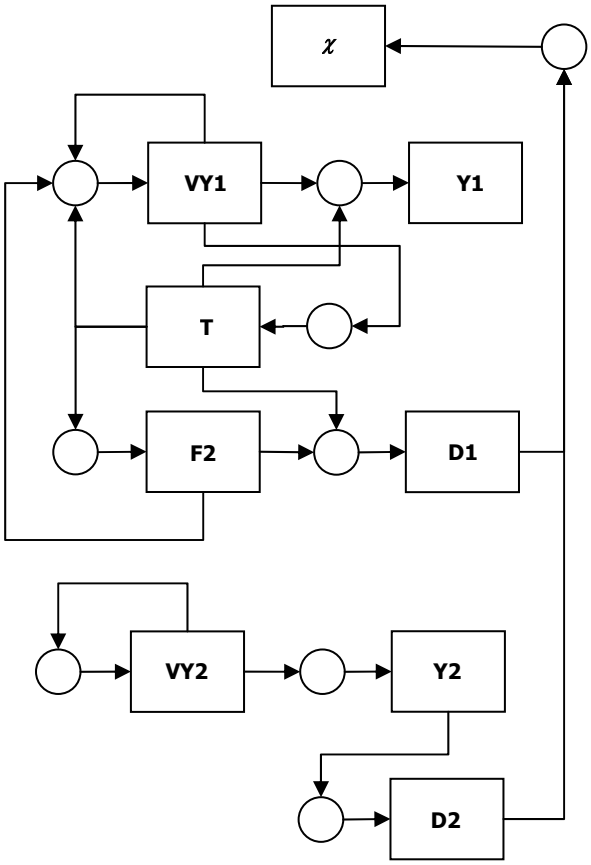

Figure 3. Rocket block diagram after separation.

When the first stage reaches the ground, the corresponding components are removed and rocket diagram becomes represented by Figure 4, where only the 2nd stage is represented.

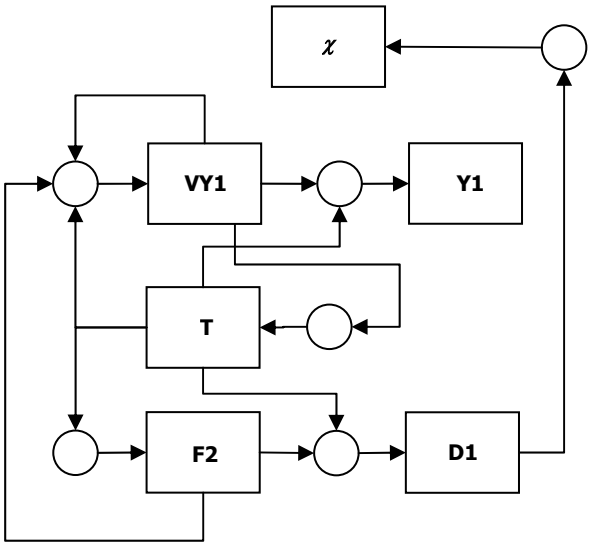

Figure 4. Rocket head block diagram.

In the case that the 1st stage reaches the ground before the 2nd stage runs out of fuel, the next event will be the depletion of the head fuel. Structural changes involve transforming the 2nd stage into a free fall body obeying to rules of gravity and friction. The new model is now described by Figure 5.

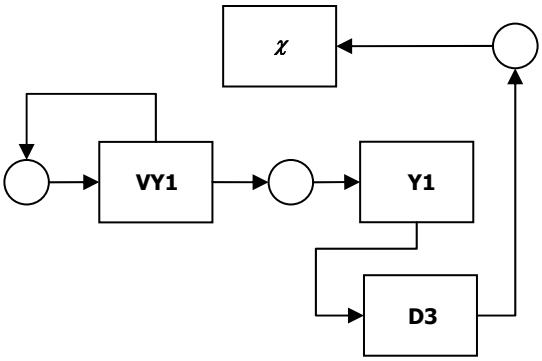

Figure 5. Free fall rocket head.

Detector #D3 will signal when rocket heads reaches the ground. This signal is sent to the network executive that will stop the simulation.

5. SIMULATION RESULTS

In this section we present the simulation results for the described system. The first structural change occurs at time 19.14 s when the 1st stage runs out of fuel. At this time two trajectories are depicted in Figure 6, one for the climbing rocked head and other for the empty tank. During the interval [0, 19.14], the rocket velocity, depicted Figure 7, has a reference value of 150 m/s. After this time a new reference velocity of 100m/s is set by the PI controller.

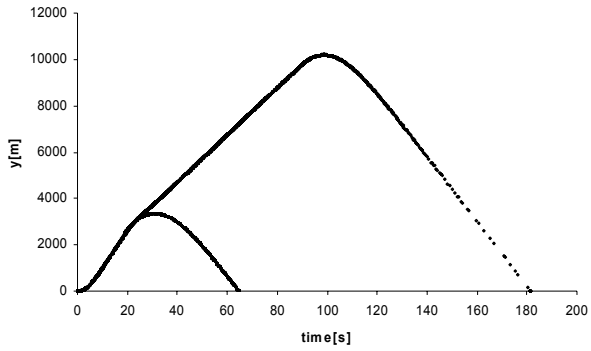

Figure 6. Position of rocket stages.

First stage reaches the ground at time 64.77 s with a velocity of 137.43 m/s. The 2nd stage continues to rise and at time 89.88 s its fuel finishes. It continues to rise and reaches a maximum height of 10187.40 m at time 98.79 s. After this time rocket head starts to fall and reaches the ground at time 181.46 s with a velocity of 140.01 m/s.

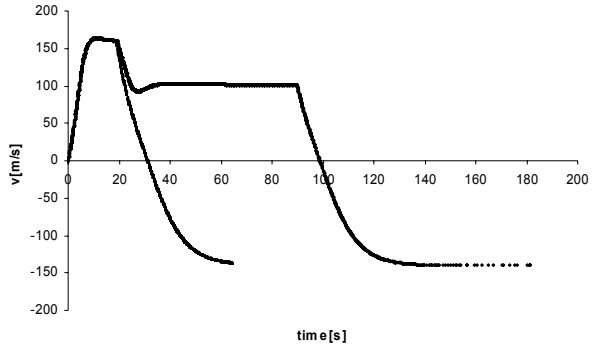

Figure 7. Velocity of rocket stages.

Fuel masses of the two tanks are depicted in Figure 8. Initial fuel mass of 1st stage is 130 Kg and its rate of consumption is proportional to the rocket thrust. After time 19.14 s Figure 8 represents the fuel of 2nd stage that has an initial value of 60 Kg.

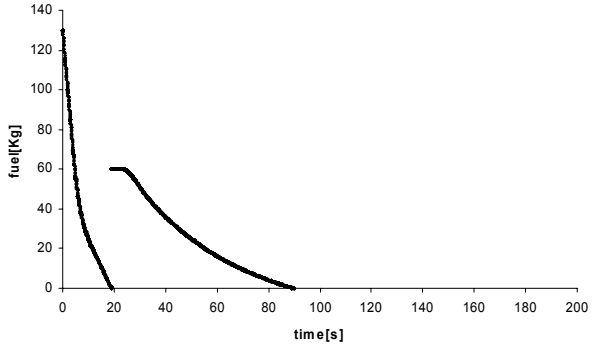

Figure 8. Fuel in rocket 1st and 2nd stages.

The thrust is set by the controller and it is depicted in Figure 9. Thrust initial value equals to 6000 N, the rocket maximum thrust, and it stays at this value during the first 5.0 s of rocket flight.

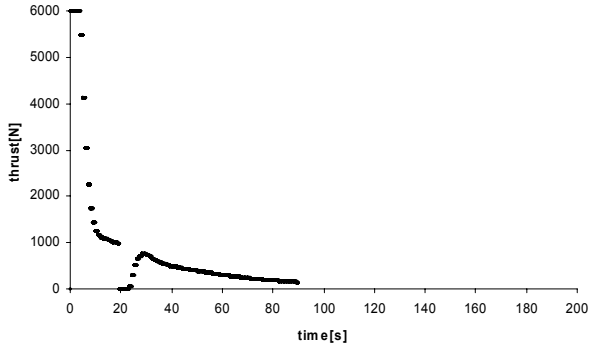

Figure 9. Rocket thrust.

When the first stage separates from the rocket head at time 19.14 s, the controller changes its parameters and the reference velocity. After time 89.88 s, when the rocket has no fuel, the controller is deactivated.

6. CONCLUSIONS

The HFSS provides a unifying formalism to represent hybrid dynamic structure models. The formalism is able to describe digital controllers, adaptive step-size numerical integrators and detectors. Giving that all these elements share the same underlying representation they can be merged and simulated together. The HFSS formalism provides a hierarchical and modular representation of systems and provides the constructs so model structure can be changed dynamically. These characteristics render HFSS an excellent framework to represent complex adaptive systems. In particular, the HFSS formalism enables the interconnection of digital controllers with different types of components.

ACKNOWLEDGEMENT

This work was partially funded by the Portuguese Science and Technology Foundation under project POSI/SRI/41601/2001.

REFERENCES

[1] Badouel, E. and Oliver, J. Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes within Workflow Systems, Research Report 3339, INRIA, 1998.

[2] Barros, F.J. Towards a Theory of Continuous Flow Models. *International Journal of General Systems*, pp. 29-39, 2002.

[3] Barros, F.J. Modeling and Simulation of Dynamic Structure Heterogeneous Flow Models. *SIMULATION: Transactions of the SCS*, Vol. 78, No. 1, pp. 18-27, 2002.

[4] Barros, F.J. Abstract Simulators for Dynamic Structure Hybrid Components. *AI, Simulation and Planning in High Autonomy Systems*, Lisbon (April 7-10), pp. 71-77, 2002.

[5] US Department of Defense. *High Level Architecture Run-Time Infrastructure Programmer's Guide*, 1998.

[6] Edström, K., J.-E. Strömberg, U. Söderman and J. Top 1997. "Modelling and Simulation of a Switched Power Converter." *3rd International Conference on Bond Graph Modeling and Simulation*, pp. 195-200.

[7] Gribaudo, M., Sereno, M., Horvath, A. and Bobbio, A. Fluid Stochastic Petri Nets augmented with Flush-Out Arcs: Modeling and Analysis. *Journal of Discrete Event Dynamic Systems*, Vol. 11, pp. 97-117, 2001.

[8] Henzinger, T.A. The Theory of Hybrid Automata, *11th Annual IEEE Symposium on Logic in Computer Science*, pp. 278-292, 1996.

[9] Zeigler, B.P. Praehofer, H. and Kim, T.G. *Theory of Modeling and Simulation*. 2nd Edition. Academic Press, 2000.