Time Windows: An Approach to Automated Abstraction of Continuous-Time Models into Discrete-Event Models

Qingsu Wang and François E. Cellier

Department of Electrical and Computer Engineering
University of Arizona, Tucson AZ 85721

ABSTRACT

This paper describes the automated generation of time windows from continuous system simulation models. Time windows can be used to automatically generate equivalent discrete–event models at a coarser granularity level, and they are also instrumental to the design of event–based control systems. The generation of time windows represents one facility in the knowledge–based multi–facetted modeling and simulation environment DEVS/Scheme. In this environment, continuous–time and discrete–event models can co–exist, and they can be amalgamated with AI techniques. The usefulness of these concepts will be demonstrated by means of a model of a robot controlled fluid handling laboratory for Space Station Freedom to be used for research in life sciences, microgravity sciences, and Space medicine.*

INTRODUCTION

Current technology has not matured to a point where it can support human life in Space in a meaningful and economic manner. For years to come, the presence of humans in Space will not be justified by a superior technology. On the contrary, it will be necessitated by an inadequate technology. Humans are needed in Space because our limited understanding of the mechanisms of intelligence do not permit us to build intelligent automated systems which can perform elaborate tasks reliably and safely under conditions of a partially unknown operating environment. However, the cost of humans in Space is high. Crew time is, and will be, a scarce and expensive resource, and consequently, science experiments in Space are frequently reduced in scope, and are often cut down to a bare minimum or below, simply because of a lack of available crew time.

The goal of our research is to develop a technology which will allow carefully designed and specially constructed laboratory robots to perform many routine tasks in a Space laboratory under remote supervision from the ground. It is thereby important to provide the robots with sufficient local intelligence so that communication with the ground can be kept within reasonable bounds, and so that the unavoidable time delays in the uplink/downlink loop do not cause any stability problems. To this end, the robots must be able to perform simple tasks autonomously, and communicate with the ground only at the task level (level 4 in the NASREM telerobot architecture [1]) and above, but never below.

However, this problem is not easy to solve. The difficulty stems from the fact that the robots must be able to operate in a partially unknown environment. Humans can co–operate in a laboratory without bumping into each other, and they can share common resources without an explicitly specified protocol of communication. If my hammer is not where I put it last, I conclude that somebody else must have taken it, and I either look for it, or I ask around who might have it. However, this requires basic problem solving skills which are not easily cast into programming language. Previous attempts at solving the collision avoidance problem and the resource sharing problem with centralized scheduling techniques are clumsy and unelegant, and the complexity of such centralized algorithms grows exponentially with the number of players involved [9].

If we wish to solve this problem elegantly and with an algorithm that grows linearly rather than exponentially with the inherent structural complexity, we must come up with a decentralized scheme in which each robot is responsible for its own doing, and operates on a limited, yet dynamically changing knowledge of its own local environment. Such an algorithm will have to be based on prediction mechanisms. The robot must be able to judge the adequacy of a proposed action plan on the basis of expectations of the effects of that action plan being executed. For this purpose, it is important that simulation models at various levels of granularity can be automatically generated at run time from a set of generic master models. Which portion of the world is included in a simulation model and which is not, and what are the levels of detail included in the model components must be decided on the basis of the dynamically changing information from the operating environment. New models will frequently be generated to be used only once and for a particular task, and they will be discarded as soon as they have fulfilled their purpose.

However, even this discussion does not unravel the entire plot. A model is useless unless it is accompanied by a matching experiment (action plan) to be performed on it. What we said about the models above, holds for the experiments as well: Experiments cannot be completely pre–designed, but they must be generated on the spot out of a set of generic master experiments.

The question now is: How do we store the knowledge of the world and its mechanisms in the generic master models and in the generic master experiments, and what are the tools

* Research supported by NASA-Ames Co-operative Agreement No. NCC 2-525, "A Simulation Environment for Laboratory Management by Robot Organizatio".

needed to automatically generate varying simulation models and accompanying simulation experiments out of these master models and master experiments? This is the realm of our research.

In this paper, we shall discuss a small subset of the overall research plan, namely the generation of models at coarse granularity levels given models developed at finer granularity levels. One such mechanism is the model pruning algorithm which was described in [11]. In model pruning, a decision is taken which variables and/or equations of the master model are to be preserved in the simulation model. All others are simply eliminated. However, model pruning does not provide for a mechanism of aggregating several variables into one, or modifying equations such that they take on a simpler form and execute more efficiently at the possible cost of a reduced accuracy. This is precisely what our new results provide us with: A tool to aggregate variables and simplify equations. Time windows are an important component of this tool.

There is yet another facet to this discussion. If we wish to create an algorithm by which a robot can operate equipment in a partially unknown environment, this algorithm had better be robust with respect to modifications of the environment. Unfortunately, the robustness and refinement of a control algorithm are usually in competition with each other. While simple proportional integral (PI) controllers work reasonably well under varying operating conditions, the more refined control algorithms, such as multivariable optimal controllers, are highly sensitive to even small variations in plant parameters [3]. Again, the problem stems from the centralized approach. If we are able to delegate control intelligence to the local subsystems while the higher level co-ordinators' responsibility is limited to ensuring the stability of their subsystems, we may be able to come up with designs that are far more robust to plant parameter variations at the expense of a slight reduction in global optimality. It is proposed that event-based qualitative controllers are good candidates for implementing such co-ordination schemes. Time windows are instrumental to the design of event-based control algorithms [2].

THE MODELING AND SIMULATION ENVIRONMENT

To satisfy the above described requirements, a modeling and simulation environment was created which comprises a five-level hierarchy of modeling tools.

At the lowest level of the hierarchy are flat simulation models, both of the discrete-event type (coded in DEVS/Scheme [17]) and of the continuous system type (coded in DESIRE [8]). In order to run efficiently, simulation models should be flat. Unfortunately, flat simulation models are hard to read and difficult to maintain.

At the next higher level of the hierarchy are hierarchically decomposed modular models of both types. Discrete-event hierarchical models are coded in DEVS/Scheme, while the continuous systems models are coded in DYMOLA [4]. Level 2 models are easier to maintain since they can reference submodels which can be stored in a model library. The transformation of level 2 models into level 1 models is accomplished by a hierarchy interpreter. In the case of the discrete-event models, the hierarchy interpreter is a built in function of the DEVS/Scheme simulation engine, whereas in the continuous case, the hierarchy is flattened by the DYMOLA preprocessor. The continuous case is a little more difficult to handle since continuous models do not provide for a natural distinction between component inputs and outputs. E.g., an electrical resistor requires the model U = R*I when connected to a current source, but it requires the model I = U/R when connected to a voltage source. The DYMOLA preprocessor contains formula manipulation algorithms which enable it to solve equations at compile time for the appropriate variable. However, level 2 models are still unwieldy since we must still code one main program for each model variant.

At the next higher level, models are represented by a pure system entity structure (SES) [10,21]. A pure system entity structure is a hierarchical tree that decomposes root entities (corresponding to the main program) graphically into its component models. The leaves of the tree correspond to level 2 atomic models, whereas the interior nodes of the tree correspond to level 2 coupled models. DEVS/Scheme contains a function (transform) which compiles a pure SES into a set of level 2 models. Atomic models are retrieved from the model library, while coupled models are automatically being generated from the information provided in the pure SES. The transform routine can generate level 2 models of either the discrete-event type (coded in DEVS/Scheme) or the continuous system type (coded in DYMOLA), and it will generate the coupled models in accordance with their use. Level 3 models are easier to create, but we still need one pure SES per variant.

At the next higher level, models are represented by general system entity structures. General SES's provide for a mechanism to describe many variants within one single SES. DEVS/Scheme provides for a tool, called ESP/Scheme [7], which can prune a general SES to generate a pure SES. In the pruning process, all variants except for one are pruned out by cutting away all undesired branches of the SES. Level 4 models are much more compact than level 3 models since we can represent an entire class of models with one single general SES. The only difficulty left is the need to decide manually which variant to keep among the many possible variants.

At the highest level of the hierarchy, we find a rule-based decision support system (DSS) which, on the basis of qualitative rules, can instruct the pruner which branches to cut. This tool is called FRASES [6].

The world model (which we previously called the set of master models and master experiments) consists of:

1) a library of level 2 leaf models and level 2 unit action plans,

2) a set of level 4 system entity structures describing the major master models, and a corresponding set of level 4 system entity structures describing the major master experiments, and

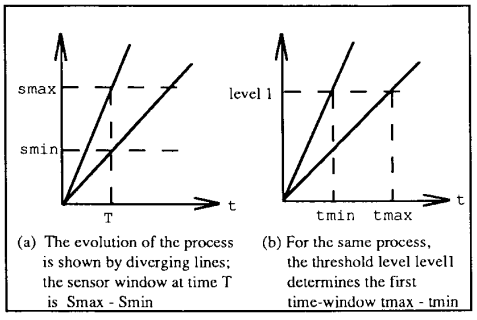
3) a level 5 rule-based decision support system which provides for the expert knowledge necessary to generate simulation models and simulation experiments for a given purpose.

The five level modeling and simulation architecture will be described in detail in a companion paper which is currently under preparation [14].

THE CONCEPT OF TIME WINDOWS IN EVENT-BASED CONTROL

Event-based control is a discrete eventistic form of control logic, in which the controller expects to receive conforming

sensory responses to its control commands within pre–set time windows which are determined by the discrete–event model of the system under control [18]. An event–based controller starts out in a *check state*, and stays in phase *wait* for the minimum allowed processing time, *tmin*. A sensory input received during this period indicates an error, since the (expected) sensory response arrived too early. Once *tmin* has elapsed without external interrupt, the model changes to phase *window*, and stays in this phase for a duration specified by the time window. A sensory input is expected to arrive during this period. If the input is received and it tests valid, an appropriate control command is issued. The *check state* is updated, and the model changes to state *wait* again for another appropriate duration, *tmin*. If the test fails, an error is reported. Finally, if the window period has elapsed without receiving the expected sensory input, another error will be processed. The event–based controller moves through its *check states* in concert with the received inputs, as long as all input signals arrive within their expected time windows.



(a) The evolution of the process is shown by diverging lines; the sensor window at time T is Smax - Smin

(b) For the same process, the threshold level level1 determines the first time-window tmax - tmin

Fig. 1 Comparison of conventional and event-based control

Contrary to the conventional sampled data logic, event–based controllers demand less precision from their sensors. These sensors can have threshold–like characteristics. The burden of the precision is placed on the time windows rather than on the sensors. Consequently, event–based controllers are less sensitive to the received sensory input, and they can therefore be made more robust. Figure 1 compares the two different logic forms of the conventional sampled data control versus the event–based control. Another important advantage of event–based control is that the error messages that are issued by the controller contain information that can be directly used for diagnostic purposes. With the event–based control paradigm, the control process can be readily interfaced with rule–based symbolic reasoning logic in an advanced robotic and intelligent automation setup.

The key to event–based control is the idea of time windows. Time windows describe the time durations from the minimum allowed process time to the maximum allowed process time. Time windows of non–zero duration are necessitated by parameter variations and by external disturbances of a process under normal operating conditions.

Time windows can be obtained from DEVS discrete–event models of the process. This feature has been discussed in [18]. However, time windows can also be directly obtained from a series of continuous system simulation runs by varying the disturbances and plant parameters of a process model in accordance with normal operating conditions. This paper introduces an approach supporting the determination of time windows directly from continuous system simulation models.

AUTOMATED ABSTRACTION OF TIME WINDOWS

DEVS/Scheme is an implementation of the DEVS formalism [15] coded in PC/Scheme [12], a LISP dialect developed for PC compatibles. DEVS/Scheme supports truly modular hierarchical model specification of discrete–event models. The simulation of discrete–event models is achieved by implementing the abstract simulator principles developed as part of the theory [15]. The ESP/Scheme software, underlying DEVS/Scheme, realizes the System Entity Structure (SES) [21] concept. DEVS/Scheme is implemented as a shell in such a way that all underlying LISP–based and object–oriented programming language features of PC/Scheme are available to the user. The result is a powerful tool for combining symbolic reasoning with hierarchical modular discrete–event modeling [17].

DEVS/Scheme and ESP/Scheme have been extended to manage continuous system models [13]. For this purpose, the continuous system models are accompanied by corresponding DEVS shell models. These models, written in the DEVS formalism, provide a knowledge level description of the continuous systems, and serve as pointers to their counterparts, the continuous system models, which are coded as sets of differential equations, and which are stored in a continuous system model base.

The modular hierarchical modeling scheme is preserved in the continuous models by using DYMOLA [4], a continuous system modeling language. DYMOLA acts as a bridge between the abstracted DEVS models and the continuous system simulation language code (DESIRE [8]). The SES is employed for the organization of all models at different levels of granularity. Management of these models is carried out by operations on the SES. Simulation trajectories can be produced to form the time information, which can then be automatically mapped into DEVS discrete–event models for future reference.

Model Bases

The knowledge base includes the following model bases.

1. ENBASE : the entity structure base. All entity structure files are in this data base.
2. MBASE: the model base. The DEVS models are stored in this data base.
3. DYMOBASE: the DYMOLA model base. It stores the DYMOLA models.
4. DEBASE: the DESIRE model base. The executable DESIRE files are saved in this data base.
5. TRAJECT: the trajectory base. The simulation trajectories are saved in this data base.

These knowledge bases are all subdirectories of a directory named DEVS.

The System Entity Structure Approach

With the existence of the atomic component models in MBASE and DYMOBASE, effective management and manipulation of these models supports their use in a variety of different system configurations. Operations performed on an SES provide means of organizing information among these model bases as well as to synthesizing, managing, and manipulating (re)usable models in the model bases.

The SES is a labeled tree with attached variables. It holds the structural knowledge of a system. An SES tree is built by construction operations, such as: create a root, add nodes, attach variables, and specify the coupling relations. Once an SES tree file has been set up, it is stored in ENBASE. DEVS models for all the leaf nodes of the SES tree are required to be resident in MBASE, and their continuous counterpart models must be resident in DYMOBASE. An operation called *prune* can be applied to an SES to generate alternatives. This pruning algorithm traverses the SES in a depth-first fashion, interactively querying the user to select one entity if there are several choices under a specialization. The querying process can be answered by the DSS instead of the user. Querying continues until all leaf entities have been visited. The result of the pruning algorithm is a pure SES. The operation *transform* can be applied only to a pure SES. Transform traverses the pure entity structure starting at the root of the tree, and calls upon a retrieve processor to search for a corresponding model of the current entity in one of the model bases. If a model has been found, the transformation of its subtree is aborted. Otherwise, the transformation continues. The *construct-continuous-systems* procedure is invoked if the model of an intermediate node cannot be found in either the working memory, the MBASE, or the ENBASE. It constructs a hierarchical DEVS coupled model, and stores it in the working memory. Simultaneously, the corresponding DYMOLA coupled model is also constructed and is saved into DYMOBASE. If the user wants to continue the transformation to obtain an executable DESIRE program, transform automatically calls upon the DYMOLA preprocessor which operates on a DYMOLA batch file that was previously generated during the transformation. DYMOLA generates an executable DESIRE program, and the transform procedure stores it automatically in DEBASE. After all these preparatory tasks have been completed, the *restart* command starts the simulation of the continuous system in DEVS, while the *run* command starts the corresponding simulation of the continuous system in DESIRE. The *run* command can be invoked from within a DEVS model. By assigning appropriate transition functions to the DEVS models, the resulting trajectories of the DESIRE simulation can be mapped into DEVS models, or they can be stored in the TRAJECT base for future reference.

This enables the user to switch back and forth between the discrete-event and the continuous system modeling concepts, exploit the advantages of both, and ensure the consistency of his modeling efforts across the barrier between the two modeling methodologies. Figure 2 visualizes the entire knowledge-based modeling and simulation environment.

Some useful macros have been created to facilitate the generation of DEVS models which incorporate the continuous simulation results. These macros operate on a pure SES. With a pure SES tree residing in ENBASE and models for all its leaf nodes residing in MBASE and DYMOBASE, these macros can be called upon to automatically transform the pruned tree, to perform series of continuous simulation runs from the tree, and to return the simulation execution times and/or the time windows. The table of parameter lists and simulation times can be saved as states of the DEVS model.

Simulation runs of a continuous system model can be performed under the control of several different experiments.

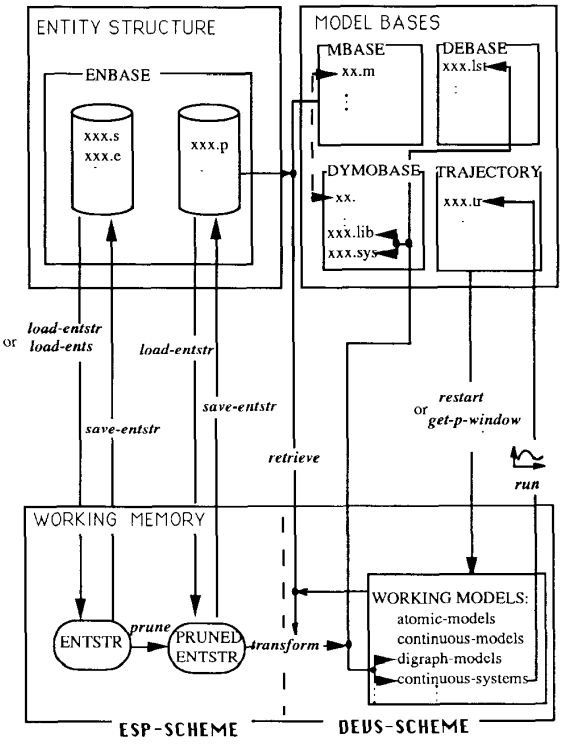


Fig. 2 The knowledge-based modeling and simulation environment

Each simulation model may have several different control models (experiments) associated with it, while one control model may be used to drive various simulation models. The simulation model and the experiment to be performed on it can be selected directly from within the DEVS macro. This enables the corresponding DEVS model to influence the execution of the continuous system simulation (coded in DESIRE).

The time windows information generated by use of the macros can then be fed back into the DEVS models to allow DEVS to perform a qualitatively similar, yet more highly aggregated, discrete-event simulation of the formerly continuous model. Such a simulation can be executed using the *restart* command once the transition functions of the atomic discrete-event models have been defined.

THE SPACE ADAPTED FLUID HANDLING SYSTEM

Our task was to design a modeling and simulation environment capable of supporting the investigation of robot organizations for managing chemical, or similar, laboratories aboard Space Station Freedom or other Space platforms. The modeling environment enables us to thoroughly study the problems to be encountered in assigning responsibilities to an organized group of robots.

Handling fluids in orbit will be essential to many experiments that are currently being planned for execution in the life sciences, microgravity sciences, and Space medicine modules. These research projects will involve many routine ma-

nipulations of fluids. Many of these experiments will not be executable without the incorporation of robot technology since sufficient crew time for manual operation will simply not be available. In the beginning, the robots assigned to such tasks will be small dedicated machines which will operate completely inside a rack, i.e., they will not at all interfere with the workspace of the astronauts. They can be viewed as movable parts of a science instrument or a set of neighboring science instruments.

In one of the considered scenarios, we envision such a robot to move along an x–z spindle in front of a set of rack–mounted instruments similar to a fork lifter in a warehouse. The robot can extract liquids from one instrument and pass it along to the next for further analysis. The robot can be considered a part of the rack, and it can be integrated into the rack in such a way that an airlock is between the instruments (including the robot) and the working area of the astronauts. In this way, we can guarantee that the safety of the astronauts will not be jeopardized by the laboratory robot.

In longer terms, our research may also be useful for automation of portions of the environmental control and life support system (ECLSS) aboard Space Station Freedom. In a closed pressurized Space environment, such as the Space Station, the growth of bacteria and funghi is inevitable. In long term missions, such as the Space Station project, this causes all kinds of health problems, such as allergies. A small robot could eventually be used to walk around in the Space Station, and analyze and remove bacterial and other contaminations, similar to a cleaning robot in a swimming pool.

In our project, a robot model consists of three parts: a motion–system, a sensory–system, and a cognition–system [20]. The cognition–system contains one selector and several Model–Plan–Units (MPUs). The selector is a controller which controls MPUs. MPUs are task specialists which are activated under appropriate circumstances [20]. For instance, one MPU may be specialized for the task of fluid handling.

The instruments considered in our current prototype setup are a pressurized bladder bottle and a syringe. Under microgravity conditions, all liquid containers must always be full. No air/liquid interfaces are allowed unless they are controlled by surface tension (e.g., in a capillary). Consequently, we cannot use standard equipment such as beakers or erlenmeiers. Instead, we use a bottle which is sealed by a septum, and which contains an inflatable bag. The air volume between the bag and the bottle walls is pressurized. Liquid is injected into or extracted from the bottle using either a syringe or a motorized pipette. The air pressure will squeeze the bag such that it remains constantly full.

To monitor and thus control the process, the robot has to have knowledge about certain states of the models of the bottle (and of the syringe). In order to model the robot's cognition of the process, several models of the same bottle are needed. These models are related to each other by abstraction [19]. In order to guarantee the integrity of the data system, it is important that all models of the bottle are automatically generated from the same master model, and that any manual modifications of the bottle perception are implemented at the level of the master model rather than directly at the level of any derived model.

Figure 3 shows that there exist three different models of the bottle: "btl-e", "btl-o',' and "btl-d". "btl-e" is the model of the bottle which is external to the MPU. "btl-e" represents

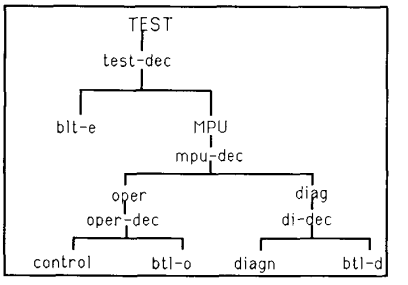


Fig. 3 SES for testing a bottle handling *mpu*

the real bottle. In a true laboratory setup, this model would be replaced by real hardware. Since the physical world is continuous, we chose to make "btl-e" a continuous system simulation model. "btl-e" is thus an abstraction of the real bottle stored in DYMOBASE. "btl-o" is the operational model of the bottle which the robot uses to control the bottle. It has been conceptualized similar to the way in which humans might view a bottle being operated on. Humans are bad at solving differential equations in their heads, and thus, "btl-o" is a much coarser model of the bottle than "btl-e". Mental models employed by humans operate on the "what–if" paradigm, i.e., humans envision a (coarse) action, and consider the (coarse) consequences of such an action being executed. This is exactly how "btl-o" works. A coarse action (i.e., a discrete event) is scheduled, and a reaction (another discrete event) is expected to occur sometime in the future, not earlier than a given minimum time, but not later than a given maximum time. "btl-o" is used by the event–based controller inside the MPU. "btl-d" is the model of the bottle used for diagnostic purposes. It mimics the way in which a human checks the states of a system when something abnormal happens. "btl-o" and "btl-d" are different abstractions of "btl-e", and as explained earlier, it is essential that "btl-o" and "btl-d" can be automatically generated either from "btl-e" directly or from a common master model through experimentation. In our setup, this is accomplished by simulation experiments, but in a real laboratory setup, this could be achieved through hardware experiments. In our scenario, time windows were automatically generated from series of continuous system simulations of "btl-e".

Notice that the above description contains a slight simplification since even "btl-e" contains a second counterpart residing in MBASE, i.e., there exists a "btl-e" DEVS model beside of the "btl-e" DYMOLA model. The "btl-e" DEVS model contains knowledge about some of the variables used in the "btl-e" DYMOLA model, and how they interrelate with the outside world. If the "btl-e" DYMOLA model were to be replaced by a real bottle, somebody would still need to tell DEVS what a bottle is, what variables can be observed, and how they fit into the rest of the world. This is the type of generic models (preconceived notions) that all humans carry around about the items that populate their everday lives. These models lack the detailed information about the specifics of a particular item. Consequently, the "btl-e" DEVS model is just a shell. It references the externally available variables of the "btl-e"

DYMOLA model, and states how they are connected with the environment of the bottle, but it does not contain any transition functions that describe the internal relations between these variables.

The simulation model of the overall fluid handling system "fh" contains, beside of the model of the physical equipment, a description of the *experimental frame* which operates on that equipment. The experimental frame [16] consists of a generator and a transducer. The robot pushes the plunger of the syringe with velocity V during the filling, and pulls the plunger with velocity $-V$ during emptying. The generator generates this input, i.e., the velocity of the system, and the transducer gathers the outputs and analyzes the results. Different generators and transducers can be used for different experimental conditions.

To simplify the modeling process, it is assumed that the input, the nominal velocity V of the syringe plunger, is constant (model "sa" is chosen). Accordingly, the nominal flow rate of the syringe is also constant. To demonstrate the flexibility of the modeling scheme of extracting time windows from continuous simulation runs which represent variations in simulation time due to parameter changes of a model, the model allows the actual flow rate into the bladder to vary in a non-linear fashion with other extraneous influencing factors. Three specific extraneous effects were considered. One was that the bladder could have a leakage. The second effect involved the angle at which the needle of the syringe penetrates the diaphragm that covers the opening of the bottle. It was thought that if this angle were very obtuse (the needle of the syringe is almost parallel to the diaphragm), the needle would not penetrate the diaphragm completely, and therefore, a fraction of the fluid ejected from the syringe would not be injected into the bottle, but would escape. The third consideration was that the flow rate of the liquid into the bottle would slow down when the bladder was almost full. These assumptions may or may not be realistic; they were included to show the ability of the modeling scheme to handle situations of this kind. For simplicity, all these effects were included in the bottle model rather than in the syringe model. The models of these effects are activated by changing the parameters of the bottle model.

GENERATING TIME WINDOWS FOR THE FLUID HANDLING SYSTEM

A system entity structure of the overall fluid handling system "fh" is stored in ENBASE (Figure 4). It can be seen

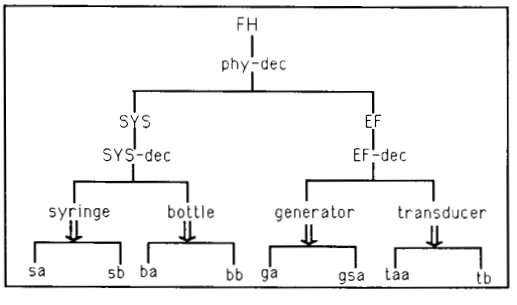


Fig. 4 The system entity structure of system fh

that all component models in the system have two specializations (of course, they could have more). For instance, the generator "ga" generates a constant input to the system while the generator "gsa" generates a sine wave input. All the eight leaf models of the SES tree are resident in MBASE and in DYMOBASE. Figure 5 shows the model *bottle* in MBASE, Figure 6 shows its counterpart model *bottle* in DYMOBASE. Figure 7 shows the pruned SES of "fh", i.e., one alternative among all the possible choices. This pruned SES is also saved in ENBASE.

```
(make-pair continuous-models 'bottle)
(send bottle valid? #t '())
(send bottle set-s (make-state 'sigma '-
                               'phase #t
                               'tflag #t
                               'tname 'bottle
                               'cut '((PPORT '(P))
                                      (IOLET '(W)))
          'parameter '((R 8.314) (M 0.00224)
                       (TEMP 273.15) (VOL 50.24)
                       (SC 1) (ANGLE 90) (LR 0))
          'local '((VOL1 0) (VOL2 '()) (FA '())
                   (FVF '()) (FVE '()) (RATE '())))
)

;-- MODEL BA --
(if (unbound? bottle)
    (load (string-append ml "bottle.m")))
(send bottle make-new 'ba)
```

Fig. 5 Model *bottle* in MBASE

```
{ MODEL BOTTLE  }
model type bottle
  cut IOLET(W /.)
  cut PPORT(P /.)
  local VOL1 VOL2
  local FA FVF FVE RATE WREAL
  { SC = 1 : Filling }
  { SC = 0 : Empty   }
  parameter R=8.314 M=0.00224 TEMP=273.15 VOL=50.24
  parameter ANGLE=90 LR=0 SC=1
  RATE = VOL1/VOL
  func FA = TAB1(ANGLE)
  func FVF = TAB2(RATE)
  func FVE = TAB3(RATE)
  WREAL = W*FA*(SC*FVF + (1-SC)*FVE) - LR
  der(VOL1) = WREAL
  VOL2 = VOL - VOL1
  P = R*M*TEMP/VOL2*1000000
end
```

Fig. 6 Model *bottle* in DYMOBASE

```
-ENT : FH
--ASP : PHY-DEC
  ,:::::coupling -> ((EF FH (OUT2 . OUT) (Y2 . Y))
                     (SYS EF (OUT . IN) ()) (EF SYS
                     (OUT1 . IN) ()))
  ,:::::variable -> ((OUTPUT Y ()))
---ENT : SYS
----ASP : SYS-DEC
  ,:::::coupling -> ((BA SYS (PPORT . OUT) (P . Y))
                     (SYS SA (IN . VPORT) (U . V))
                     (SA BA (IOLET . IOLET) ()))
  ,:::::variable -> ((CUT OUT (Y / .)) (CUT IN (U / .)))
-----ENT : SA
-----ENT : BA
---ENT : EF
----ASP : EF-DEC
  ,:::::coupling -> ((TAA EF (OUT . OUT2) (Y . Y2)) (GA
                     EF (OUT . OUT1) (Y . Y1)) (EF TAA
                     (IN . IN) (U . U)))
  ,:::::variable -> ((CUT OUT2 (Y2 / .)) (CUT OUT1
                          (Y1 / .)) (CUT IN (U / .)))
-----ENT : GA
-----ENT : TAA
end of display
```

Fig. 7 Screen output of the pruned SES of fh

In this paper, we only discuss the filling process. Generating time windows for emptying is just a matter of changing the initial states of the components and the input of the system.

```
cmodel
simutime 10
step 0.01
commupoints 100

ctblock
connect 'fh.tr' as output 2
dimension TAB1[12],TAB2[10],TAB3[10]
data 0, 25, 45,  80,  85, 90
data 0, 0,  0.2, 0.8, 1,  1
data 0, 0.9, 0.95, 0.99, 1
data 1, 1,   0.9,  0.1,  0
data 0, 0.01, 0.05, 0.1, 1
data 0, 0.1, 0.9, 1, 1
read TAB1
read TAB2
read TAB3
dimension lra[5]
data 0, 0.1, 0.04, 0.06, 0.08
read lra
dimension angle[5]
angle[1] = 90
angle[2] = 65
for i=3 to 5
 angle[i]=abs(ran(0))*30+70
next
for i=1 to 5
 ANGLE=angle[i]
 LR=lra[i]
 drun
 write #2,ANGLE,LR,t
 reset
next
disconnect 2
ctend

outblock
OUT
```

Fig. 8 Simulation control model $fh.1$

In the DYMOLA model "bottle", it is shown that the ports of the bottle are declared as *cuts* [4]. Variable *VOL1* denotes the volume of the bladder, and *VOL2* denotes the volume between the bladder and the wall of the bottle. *VOL*, a constant, denotes the total volume of the bottle. Fluid can flow in or out through the port *IOLET*. The input/output variable at port *IOLET* is the nominal flow rate W. The actual flow rate *WREAL* is influenced by the factors mentioned above. The effect of the injection angle on the flow rate is described as a tabular function *TAB1*, and that of the fluid volume in the bladder is a tabular function *TAB2* for filling, and another tabular function *TAB3* for emptying. The effect of leakage is described through the variable *LR*. Values for these tabular functions are declared in the simulation control model (Figure 8). According to the gas law, the variable P, i.e., pressure, at port *PPORT* indicates the pressure between the bladder and the wall which is related to the volume of the fluid in the bladder [5]. The generator "gen" generates the nominal velocity of the syringe plunger. The nominal flow rate of the syringe is the product of its cross-sectional area and the velocity of its plunger. The transducer "transdu" in our example simply rescales the input variable, i.e., the pressure P.

A more detailed study of these models could be performed. Modifications can be made to the individual models, and different types of models can be chosen without changing the system entity structure.

With the existence of a pruned entity structure in ENBASE

```
;-- Get filling time window

(define (example-1)
; loading the model into working memory
   (if (unbound? ba)
       (load (string-append ml "ba.m")))
   (if (unbound? sa)
       (load (string-append ml "sa.m")))
   (if (unbound? ga)
       (load (string-append ml "ga.m")))
; for filling the generator generates V > 0
   (send ga change-parameter '((SC 1.5)))
; set the initial volume of liquid in syringe full
   (send sa change-ic '((VOL 50.24)))
; set the parameter SC of bottle to be 1 for filling
   (send ba change-parameter '((SC 1)))

; set the initial condition for VOL1 of bottle to
  be zero and get time window
   (eval '(get-p-window p:fh-a ba '((vol1 0))
                       '(ad lr) 1 5 "fh$Y-2.14E+6")))
```

Fig. 9 Procedure to obtain the filling time window

```
[2] (example-1)

(FH ROOT-ASP)
(SYS PHY-DEC FH ROOT-ASP)
(SA SYS-DEC SYS PHY-DEC FH ROOT-ASP)
(BA SYS-DEC SYS PHY-DEC FH ROOT-ASP)
(EF PHY-DEC FH ROOT-ASP)
(GA EF-DEC EF PHY-DEC FH ROOT-ASP)
(TAA EF-DEC EF PHY-DEC FH ROOT-ASP)
-- Do you want to continue the transformation of
-- the models to get the executable continuous
-- system simulation files? (y/n)
y

=============
root-co-ordinator: R:FH
-model: FH---> processor: C:FH
--model: SYS---> processor: C:SYS
---model: SA---> processor: S:SA
---model: BA---> processor: S:BA
--model: EF---> processor: C:EF
---model: GA---> processor: S:GA
---model: TAA---> processor: S:TAA
=============

-- Do you want to save another trajectory
-- besides the basic one ?
   (y/n)
n

(3.64 6.97)

[3] (send ba get-sv 'p-table)

    (((ANGLE 9.00000E+01) (LR 0.00000E+00) 3.64000E+00)
     ((ANGLE 6.50000E+01) (LR 0.1) 6.97000E+00)
     ((ANGLE 7.00000E+01) (LR 0.04) 5.86000E+00)
     ((ANGLE 7.00010E+01) (LR 0.06) 5.96000E+00)
     ((ANGLE 7.66445E+01) (LR 0.08) 5.06000E+00)
    )
```

Fig. 10 Result from extracting the filling time window

and the component models in MBASE and DYMOBASE, DEVS macros can be used to execute continuous simulation runs and to obtain the desired time trajectories.

In this application, different parameter values within the range of normal operating conditions were assigned to the model "syringe" and the model "bottle". Time windows are then determined by the maximum and minimum simulation times recorded for various values of a model parameter.

To make the simulations more efficient, i.e., save the time needed for the transformation of the pruned SES, the parameter changes were specified directly in the simulation control

model rather than as attached variables in the SES. When calling the macro *get-p-window*, a test number is specified to indicate the particular simulation control model to be used.

Figure 9 shows that macro *get-p-window* sets the initial volume of the bladder to zero, it sets the test number to 1, and it executes five separate simulation runs with different parameter values. The simulations are terminated when the system output reaches a value of $2.14 \cdot 10^6$. The system output is the output of "transdu", which is the rescaled pressure of bottle "ba". The value $2.14 \cdot 10^6$ is the rescaled pressure indicates that the bladder in "ba" is full.

Run "(example-1)" calls upon the macro. The macro transforms the pure SES, performs the required simulations, and finally returns the desired time windows. The result is shown in Figure 10. To get the parameter values of the model and the simulation time for every simulation, method *get-sv* can be sent to model bottle. Besides from returning the time windows, the macro also produces the continuous model files, the simulation program, and the trajectory files in DYMOBASE, DEBASE, and TRAJECT.

CONCLUSION

This paper presents an approach to automatically generating time windows for intelligent event–based control in a knowledge–based modeling and simulation environment. Our approach has been exemplified at hand of a robot controlled fluid handling system designed for Space Station Freedom. Other methods of automatically mapping continuous–time models into equivalent discrete–event models can be readily exploited in our modeling and simulation environment. Thereby, continuous process control can be interfaced with a symbolic reasoning system. Continuous system models and discrete–event models can co–exist in the environment, and they can be amalgamated with more classical AI techniques such as rule–based expert systems.

REFERENCES

[1] Albus, J.S., H.G. McCain, and R. Lumia (1987), *NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)*, NBS Technical Note 1235, Robot Systems Division, Center for Manufacturing Engineering, National Technical Information Service, Gaithersburg, MD.

[2] Chi, S.D., and B.P. Zeigler (1990), "DEVS–based Intelligent Control of Space Adapted Mixing Process", (accepted for presentation at *Fifth Conference on Artificial Intelligence for Space Applications*, Huntsville, AL).

[3] D'Azzo, J., and C.H. Houpis (1988), *Linear Control System Analysis and Design: Conventional and Modern – 3rd Ed.*, McGraw Hill, New York.

[4] Elmqvist, H. (1978), *A Structured Model Language for Large Continuous Systems*, Ph.D. Dissertation, Report CODEN: LUTFD2/(TFRT-1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[5] Franks, R.G.E. (1972) *Modelling and Simulation in Chemical Engineering*, John Wiley and Sons, Inc., New York.

[6] Hu, J., J.W. Rozenblit, and Y.-M. Huang (1989), "FRASES — A Knowledge Representation Scheme for Engineering Design", *Proc. SCS MultiConference on Advances in A.I. and Simulation*, Tampa, FL, pp. 141–146.

[7] Kim, T.G. (1988), *A Knowledge–Based Environment for Hierarchical Modelling and Simulation*, Ph.D Dissertation., Dept. of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721.

[8] Korn, G.A. (1989), *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

[9] O'Donnell, P.A., and T. Lozano–Perez (1989), "Deadlock–Free and Collision–Free Coordination of Two Robot Manipulators", *Proc. IEEE International Conference on Robotics and Automation, Vol. 1*, Scottsdale, AZ, pp. 484–489.

[10] Rozenblit, J.W., *et al* (1989), "An Integrated, Entity–Based Knowledge Representation Scheme for System Design", *Proc. of NSF Engineering Design Research Conference*, Amherst, pp. 393–408.

[11] Rozenblit, J.W., and Y. Huang (1989), "Rule–Based Generation of Model Structures in Multifacetted Modeling and System Design, *ORSA Journal on Computing*, (in review).

[12] Texas Instrument (1985), *TI Scheme Language Reference Manual*, Dallas, TX.

[13] Wang, Q. (1989), *Management of Continuous System Models in DEVS–Scheme: Time Windows for Event Based Control*, MS Thesis, Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721.

[14] Wang, Q., F.E. Cellier, and B.P. Zeigler (1990), "A Five–Level Hierarchy for the Management of Simulation Models", in preparation.

[15] Zeigler, B.P. (1984), *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London, U.K. and Orlando, FL.

[16] Zeigler, B.P. (1985), *Theory of Modelling and Simulation*, John Wiley, New York, 1976; reissued by Krieger, Malabar, FL.

[17] Zeigler, B.P. (1987), "Hierarchical, Modular Discrete–Event Modelling in an Object–Oriented Environment", *Simulation*, **50**(5), pp. 219–230.

[18] Zeigler, B.P. (1989), "DEVS Representation of Dynamical Systems: Event–Based Intelligent Control," *Proceedings of the IEEE*, **77**(1), pp. 72–80.

[19] Zeigler, B.P. (1990), *Object-Oriented Simulation with Hierarchical Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, Orlando, FL, and London, U.K.

[20] Zeigler, B.P., F.E. Cellier, and J.W. Rozenblit, (1988), "Design of a Simulation Environment for Laboratory Management by Robot Organizations", *J. Intelligent and Robotic Systems*, **1**, pp. 299–309.

[21] Zeigler, B.P., and G. Zhang (1988), "The System Entity Structure: Knowledge Representation for Simulation Modelling and Design", *Artificial Intelligence, Modelling and Simulation*, (L.E. Widman *et al.*, eds.), John Wiley and Sons, New York.