June 1999

# Semi-Qualitative Simulation for Virtual Environments

Charles A. Erignac
*University of Pennsylvania*



Postprint version. Published in *The Thirteenth International Workshop on Qualitative Reasoning*, June 1999, 12 pages.
Publisher URL: http://portal.acm.org/citation.cfm?id=933336&dl=&coll=&CFID=15151515&CFTOKEN=6184618

# Semi-Qualitative Simulation for Virtual Environments

**Abstract**
We present an object-oriented semi-qualitative modeling language and a web-based simulator implementation. This language is used to develop physics based simulations in virtual environments. The associated simulator is a self-contained module. It produces, in parallel, numerical data for interactive visualization and qualitative data. The latter is available to any autonomous agents inhabiting the environment. This work is part of an ongoing project to develop self- explanatory maintenance procedure simulation in virtual environments.

**Comments**
Postprint version. Published in *The Thirteenth International Workshop on Qualitative Reasoning*, June 1999, 12 pages.
Publisher URL: http://portal.acm.org/citation.cfm?id=933336&dl=&coll=&CFID=15151515&CFTOKEN=6184618

# Semi-Qualitative Simulation for Virtual Environments

**Charles A. Erignac**
Center for Human Modeling and Simulation
University of Pennsylvania, PA 19104-6389, USA
cerignac@graphics.cis.upenn.edu

### Abstract

We present an object-oriented semi-qualitative modeling language and a web-based simulator implementation. This language is used to develop physics based simulations in virtual environments. The associated simulator is a self-contained module. It produces, in parallel, numerical data for interactive visualization and qualitative data. The latter is available to any autonomous agents inhabiting the environment. This work is part of an ongoing project to develop self-explanatory maintenance procedure simulation in virtual environments.

## Introduction

Simulations in virtual environments are becoming increasingly common. However, few of them provide monitoring, self-explanatory, or tutoring functions. Some progress has been made with systems like the STEVE (Johnson & Rickel 1997) training environment. The embodied agent STEVE can explain and demonstrate the operation of virtual machinery. This system focuses on producing realistic verbal and non-verbal communication between tutor and trainee. We are also engaged in the field of self-explanatory simulation involving virtual humans. While STEVE uses preset causal relationships to explain physical behaviors, we are trying to infer causality from actual simulation. In a broader perspective, we are working to fill the gap between physical simulation in virtual environments and autonomous agents.

In this paper, we will present a semi-qualitative modeling language and its simulator. These components are part of an on-going project to simulate maintenance procedures on hydraulic systems (Badler & Erignac 1998)

## Motivation

We are designing a language to support the concepts of *compositional modeling* (Falkenhainer & Forbus 1991). In this paradigm, physical devices are built out of elementary blocks named *model fragments*. With careful modeling one can author libraries of fragments for different physical domains and combine them at will. This method allows one to make modeling assumptions explicit, by stating them in the simulation scenario. A user can refer to them later to assess the simulation's domain of validity. These assumptions can also trigger a set of rules to select ancillary or interdependent models.

The Compositional Modeling Language (CML) (Falkenhainer *et al.* 1994) and its extension, the Compositional Modeling Interchange Language (CMIL) (Iwasaki *et al.* 1997) were the first attempts to define a standard qualitative modeling language. They are used in the Collaborative Device Modeling Environment (CDME) (Iwasaki *et al.* 1997). CDME enables engineers to design and test device models through a web-based interface. This system uses a suite of Lisp programs to run semi-qualitative simulations. The semi-qualitative simulators Pika (Franz G. Amador & Weld 1993) and SIMGEN (Forbus & Falkenhainer 1990a; 1990b; 1995) also use Lisp-based modeling languages.

We chose not to use any of these languages for two reasons. First, the simulator was to be implemented in C++ for speed and ease of interfacing with a commercial VR system like Jack (EAI/Transom ). Second, we wanted a syntax with the constructs, look and feel of regular object-oriented languages (i.e.: attributes, methods, encapsulation, inheritance).

One could argue that these reasons are purely subjective and that Lisp, which is deeply rooted in the AI community, would perform as well. One of our goals is to expose a wide audience of object-oriented developers to the concepts of qualitative physics. The best way to do so is to use their dialects.

This choice raises the interesting problem of selecting a qualitative behavior streaming format which enables dialogs between the simulator and Lisp-based agents. Nevertheless, the Lisp-like syntax is very appropriate for defining patterns and relations. We used a similar syntax for the internal representation of the logical

elements of our language.

We bind model fragments with *influences* (Forbus 1984a). However, our approach is different from the classic *direct/indirect* treatment. It is up to each fragment to filter and *resolve* (Forbus 1984a; Kuipers 1994) explicitly its influences. This is similar to the formulae used to resolve forces in classical physics.

Physical processes are the key components of qualitative physics (Forbus 1984a). With them, one can model entire physical domains from *first principles*. They model the flows of matter or energy that spontaneously occur in physical systems. Their explicit modeling simplifies explaining physical behaviors in qualitative terms. They are supported by the language as *self-instantiating* fragments.

The numerical models built by semi-qualitative simulators are dynamic hybrid-systems (Alur *et al.* 1994; Mosterman & Biswas 1997). There are a large number of hybrid systems modeling languages and associated simulators available. These programs produce only quantitative data. Some of them can instantiate model classes during the simulation (Deshpande, Göllü, & Semenzato ), but they do not have inference engines to trigger instantiation rules of qualitative physics. Also, there is no support for influence resolution. However, objects are generally modeled as finite state machines. This feature allows to model complex components easily. Likewise, our language has state and transition constructs to embed finite automata in model fragments.

We chose to support lineal differential algebraic equations as a compromise between speed and expressiveness.

## Modeling Language

In this section we present the main concepts of our modeling language. After a brief definition of primitive types, we introduce the language's dual representation. This key feature enables logic and procedural programming to cohabit within an object-oriented framework. Then, we show how model fragments relate to object classes. The following three subsections present our quantity model, the expressions and statements. We complete our overview with our treatment of constraints, influence resolution and self-instantiation. We will, along the way, define a `sigma` arithmetic operator on which our model composability depends.

### Primitive Types

The language has simple primitive types and object primitive types. The simple types are booleans, integers, reals, and predicates. Predicates are boolean attributes of fragment instances.

The primitive object types are:

**Quantities** A quantity is a semi-qualitative state variable. It is equivalent to a real variable.

**States** A state has a specific name and can contain logical, algebraic, and differential constraints. Every fragment has a default `start` state

**Transitions** A transition is defined between two states and is guarded by a boolean expression. An optional block of statements is executed each time the transition is performed.

### Dual Representation

Qualitative modeling uses first order predicate calculus to implement the high-level logical control of model fragments (Forbus 1984a; de Kleer & Brown 1984; Falkenhainer *et al.* 1994). However, object-oriented languages use a different operational semantic. Therefore, we use a dual representation for each fragment instance, quantity, predicate, transition, and state.

In the first part of the representation, these objects are entries in the procedural execution environment (symbol table, heap and stack) (Aho, Sethi, & Ullman 1985). In the second, they are facts of a global knowledge base (KB). Each fact is identified by a unique Lisp-like term called *tag*.

The procedural execution context supports the object-oriented constructs of the language.

The logical constructs, such as rules, transitions, or instantiation conditions are managed by a pattern-directed inference engine (Forbus & de Kleer 1993). It uses the KB to draw inferences and records them in a Truth Maintenance System (TMS) (Forbus & de Kleer 1993).

A type is *fact equivalent* if all of its instances are associated with a fact in the KB. By extension, an instance is fact equivalent if its type is fact equivalent.

A fact is either unknown, true or false. A predicate has the same value as its corresponding fact. The predicate `P` of an instance `x` is referred by `x.P` in object-oriented notation will have the tag `P(x)` in the KB. For the rest of this paper we will mean by predicate a fact in the KB. This includes predicate attributes since they are "unary predicates". We will make the distinction if necessary.

Given an instance `x` and one of its properties `y` of primitive object type (quantity, state and transition), the tag of `y` is `(y x)`.

The fact representing a fragment instance, a quantity, or a state is true if the object exists. It is unknown otherwise. For the remainder of this paper we will consider an object, its tag, and representing fact as one entity. So saying that an object is true means that it exists. If it is unknown, it has been retired or was never instantiated.

Every object instance has a default predicate named `Active`. It indicates whether the instance is active. All objects, except states and transitions, are always active as long as they exist. Fragment instances that have states are initialized in their `start` state when they become active.

## Model Fragments

Model fragments are the programming units of the language. They are similar to object-oriented classes. Their attributes can be of primitive type or references to other fragment instances. An instantiation clause renders a fragment self-instantiating. So far, the language does not support inheritance.

The body of a fragment defines a local naming context, an execution context, and a partial constraint environment.

The quantities, states, and transitions of an instance are created during the instantiation of the fragment itself. They share the instance's life span. The quantities are active if and only if the instance is active. Only one state is active at a time. Similarly, there is at most one active transition within the scope of a fragment instance.

A model fragment can also feature *methods* (user defined functions). The keyword `create` is reserved for identifying the *Eiffel-style* (Meyer 1991) constructor of a model fragment. This function returns a new instance of the fragment each time it is applied to a variable of the same type. It binds the variable with the new instance as a side effect.

So far, the language has no provisions for garbage collection. Objects are retired by retracting them (i.e.: setting their corresponding fact to unknown). They remain allocated but without logical existence. An object can stay in limbo until it is reasserted (i.e.: its corresponding fact is set to true). This feature is particularly useful for implementing intermittent processes.

## Quantities

As stated earlier, quantities are semi-qualitative state variables. Each of them has a user defined set of *landmarks* (Kuipers 1994). A landmark is defined by an arithmetic expression which is evaluated once when the quantity is instantiated. The qualitative magnitude of a quantity is updated according to its numerical value. The qualitative derivative is not yet supported by the language. It can be computed if the quantity has an explicit derivative.

A quantity has three built-in predicates: `IsConst`, `IsUnknown`, and `IsIntegral`. They are mutually exclusive and indicate whether the quantity is a constant, an unknown[1] of the global differential system, or the integral of another quantity (see differential constraint).

## Expressions

The syntax of the language allows for logical and arithmetic expressions. *Pure* Boolean expression can be built with the operators `and`, `or`, `not`, `=>` and `<=>`. They can contain references to fact equivalent objects. They can also refer simple predicates (facts that do not represent objects) in a Prolog-like form[2].

The arithmetic expressions are built with constants, real or integer variables, quantities, and the operators : `+`, `-`, `*`, `/`, `^` , `sign`, and `sigma`. The definition of the `sign` operator is $sign(x) = if\ x = 0\ then\ 0\ else\ x/|x|$. The `sigma` operator will be defined later.

*Mixed* boolean expressions are build by nesting arithmetic expressions with the relational operators `=`, `<>`, `>`, `<`, `>=`, and `<=`.

An *arbitrary* boolean expression can be pure or mixed.

## Statements

So far the language has only three types of statements: assignment, assertion, and retraction. They can be inserted in the body of functions or transitions.

The assignment *`expr1 := expr2 ;`* stores the value of *expr2* at the reference returned by *expr1*. Example: `plane.weight := plane.mass*g;`.

The statements `Assert(`*P*`);` and `Retract(`*P*`);` respectively assert and retract the predicate *P*. Example: `Assert(FlowOf(a,t));` where a and t are fragment instances.

Further development of the language should add flow control statements to encode more complex procedural behaviors during transitions.

## Constraints

Logical, algebraic and differential constraints define the logical, continuous and dynamic behaviors of a model fragment. They are defined within the body of a fragment or in one of its states. When a fragment or a state is instantiated, the constraints of its body are applied within the context of the created instance. These applied constraints form a partial constraint environment. The latter is active when the instance it belongs to is active. This means that the constraints defined in a state are active if and only if the state is active. The

[1] Unless specified, an unknown quantity `x` means that `x` is an unknown and not that its equivalent fact is unknown (i.e.: retracted).

[2] A Lisp-like syntax is used for internal representation of KB tags. The Prolog form is used in the expressions because it is similar to function calls and fits better in the overall syntax.

total constraint environment is the union of all the active partial constraint environments. It is used to model the behavior of the whole physical system.

**Logical Constraints** The language supports clauses and rules (Forbus & de Kleer 1993).

A clause is defined by an arbitrary boolean expression. The completeness of the inferences that can be drawn from a set of active clauses depends on the implementation of the simulator.

Example: given the model fragment `AirplaineSeat` with the predicates `Up_Right`, `Tray_Stowed` and `Safe`; given the `AirplaineSeat` instance `seat`, the constraint

```
clause seat.Up_Right and seat.Tray_Stowed
  <=>seat.Safe;
```

is equivalent to
$seat.Up\_Right\ and\ seat.Tray\_Stowed \Leftrightarrow seat.Safe$.

A rule definition contains a *context*, an *operator*, a *trigger* and a *body*. They are defined as followed:

**Context** A declaration of local variables of fact equivalent type to be used in the trigger and body. It is equivalent to the local context of a procedure.

**Trigger** An arbitrary boolean expression.

**Body** A sequence of Prolog-style predicates.

**Operator** The token `=>` or `->`. The double arrow `=>` is equivalent to $\Rightarrow$. The first time the rule is triggered, the condition of the trigger becomes an antecedent of the aforementioned predicates. The single arrow `->` simply asserts the predicates when the rule is triggered.

Each time a rule of the form *`context:body => predicate`* fires, it is translated into a clause *context* $\wedge$ *body* $\Rightarrow$ *predicate* after substitution of the local variables in *`context`*, *`condition`* and *`predicate`*.

Rules of the form *`context:body -> predicate`* are procedural. When they fire, their predicates are asserted after substitution of the context. Unless they are explicitly retracted, these assertions remain even if the firing conditions cease to hold.

Example: let `Dog` be a fragment model with the quantity `age` as property. The constraint

```
rule Dog d:(d.age<=2.0)=>Is_A_Puppy(d);
```

is equivalent to $\forall d \in Dogs, d.age \leq 2 \Rightarrow Is\_A\_Puppy(d)$ where $Dogs$ is the set of all `Dog` instances.

Unlike rules, clauses do not impose a causal ordering.

**Algebraic Constraints** Algebraic constraints are implemented as linear algebraic equations. The construct `equation` *`exp1=exp2`*`;` defines the algebraic equation $exp1 = exp2$ where *`exp1`* and *`exp2`* are arithmetic expressions. They must be linear for the unknown quantities they contain.

Equations are the real equivalent of clauses. They do not impose a causal ordering.

**Differential Constraint** The language has a differential constraint that binds one quantity as the time derivative of the other. For example `derive x,y;` means $y = \frac{dx}{dt}$.

Given `derive x,y;`, `x` is the integral of `y` and `y` is the derivative of `x`. At any time a quantity can have at most one integral and one derivative.

Upon activation the constraint `derive x,y;` will assert the `IsIntegral` predicate of `x`. When it gets deactivated it asserts the `IsUnknown` predicate of `x`. This indicates that `x` is not an integral anymore and has to be solved.

The `derive` constraint is similar to those found in numerical simulation languages (Cellier 1991). However it goes against the notion of additivity of direct influences (Forbus 1984a). We will show how the language remedies this situation, but first we need to define a special operator.

## `sigma` Operator

The `sigma` operator has three components: a context, a trigger, and an arithmetic expression. The context and trigger have the same meaning as in a rule. The expression is within the naming domain of the context. Like in a rule the trigger defines a set of substitutions. When we apply a substitution to the expression we get a new expression similar to the original except that the references have been substituted.

To evaluate a `sigma` operator we compute the valid set of context substitutions (at the time of the evaluation),*expand* the expression with the set and evaluate the result. To expand means to apply each substitution to the expression and sum all the substituted expressions. This operation produces the expression symbolically equivalent to `sigma` operator for a given substitution set.

More formally, given the global variables $x_1, \ldots, x_n$ the context $T_1\ y_1, \ldots, T_m\ y_m$ where $y_1, \ldots, y_m$ are the local variables of respective type $T_1, \ldots, T_m$;
given the boolean and arithmetic expressions
$trig(x_1, \ldots, x_n, y_1, \ldots, y_m)$
and $expr(x_1, \ldots, x_n, y_1, \ldots, y_m)$;
then
`sigma(`$T_1\ y_1, \ldots, T_m\ y_m$`:`$trig(x_1, \ldots, x_n, y_1, \ldots, y_m)$`|`
$expr(x_1, \ldots, x_n, y_1, \ldots, y_m)$`);`

equals

$$\sum expr(x_1, \ldots, x_n, y_1, \ldots, y_m)$$
$$\forall y_1 \in T_1, \ldots, y_m \in T_m : trig(x_1, \ldots, x_n, y_1, \ldots, y_m)$$

The `sigma` operators featured in statements are evaluated according to the formula above. Those used in active `equation` constraints are expanded. Their substitution set is continuously updated. Each time set changes the corresponding expansion is regenerated.

Example: let `t` be an instance of the model fragment `Tank`. Let `a`, `b` and `c` be three instances of the `Flow` model fragment. `Tank` has three quantities `inFlow`, `outFlow` and `netFlow`. `Flow` has one quantity `q`. Let `FlowOf(f,t)` be the relation indicating that *f* is a flow of *t*. This relation holds as long as both instances are active.

`Tank` has the following constraints:

```
equation inFlow=sigma(Flow f: FlowOf(f,self)
 and f.q>0.0| q);
equation outFlow=sigma(Flow f: FlowOf(f,self)
 and f.q<0.0| q);
equation netFlow=inFlow-outFlow;
```

Let us assume that at a given point of the simulation: `FlowOf(a,t)`, `FlowOf(b,t)`, `FlowOf(c,t)`, `a.q`$>0$, `b.q`$<0$, and `c.q`$>0$.

Then the two first equations are expanded as $inFlow = a.q + c.q$ and $outFlow = b.q$.

If after a certain time, the flow `a` ceases to exist and `c.q`$<0$ then $inFlow = 0$ and $outFlow = b.q + c.q$.

## Influence Resolution

The Qualitative Process Theory (QP Theory) (Forbus 1984b) states that given two quantities $x_i$ and $y$, $x_i$ influences $y$ if $y$ is functionally dependent on $x_i$. An influence is either direct or indirect.

The indirect influence $Q^+(y, x_i)$ means that there exists an $f$ such that $y = f(\ldots, x_i, \ldots)$ and $\frac{df}{dx_i} > 0$.

The direct influence $I^+(y, x_i)$ means $\frac{dy}{dt} = sum(\ldots, x_i, \ldots)$[3].

The actual "assembly" of model fragments is accomplished by *resolving* the influence of each quantity. To do so, one converts all the influences on a quantity into a constraint. Before doing so, one must assume knowledge of all the influences on the given quantity. This assumption, known as *closed world assumption*, must be revised each time an influence is added or removed.

The definition of `derive` states that a quantity can be directly influenced only once at a time (i.e.: direct influences are exclusive). This means that direct influences do not need to be resolved. However, we need to find a way to express multiple direct influences.

In QP theory, direct influence resolution is done as follows: given a quantity $I^{s_i}(y, x_i)$ where $\forall i \in [1, n]\ s_i \in \{-, +\}$ then $y' = ssum(s_1, \ldots, s_n, x_1, \ldots, x_n)$ where $y' = \frac{dy}{dt}$. *ssum* is a sum in sign algebra such that if $s_i = +$ then $\frac{\partial y'}{\partial x_i} = 1$ otherwise $\frac{\partial y'}{\partial x_i} = -1$.

[3] These definitions are from (Kuipers 1994).

Since we cannot express multiple direct influences on $y$, let us convert the direct influences $I^{s_i}(y, x_i)$ where $i \in [1, n]$ into the indirect ones $Q^{s_i}(y', x_i)$.

The QP theory solution to the indirect influences is $y' = M(s_1, \ldots, s_n, x_1, \ldots, x_n)$. This means that $y'$ is a monotonic function of $x_i$ and if $s_i = +$ then $\frac{\partial y'}{\partial x_i} > 0$ otherwise $\frac{\partial y'}{\partial x_i} < 0$.

The monotonic constraint $M$ represents a large class of functions. However, in classical physics, influences are flows of matter or energy. Their resolution is formulated by conservation laws which have the form of sums. As a consequence, we will use a sum instead of $M$[4].

This simplification complies with the qualitative definition of direct influence resolution. On the other hand, it restricts the expressiveness of indirect influences. We trust that this will not hinder the modeling capacity of the language.

Finally, the method to express direct and indirect influences is to use `sigma` operators. This is done in two steps:

**Step1** Create a predicate which will bind the fragment that contains the influenced quantity with the fragment of the influencing quantity (or the quantity itself). The name of the predicate can be used to classify the nature of the influence.

**Step2** In the influenced fragment, write an explicit resolution formula with an `equation` featuring the influenced quantity and a `sigma`. The context should contain a variable whose type is the influencing fragment. The trigger should be a condition to match the binding predicate.

The flow conservation example we gave earlier (see `sigma` Operator) uses the predicate `FlowOf(f,t)`. The latter indicates that the flow *f* influences the `inFlow` or `outFlow` of the tank *t*. The two first equations in the `Tank` fragment resolve the `FlowOf` influences with additional constraints (direction of flow).

The example also illustrates the ability of `sigma` to update itself when the closed world assumptions change. In this case, the change was due to the termination of flow `a` and the reversal of flow `c`.

The explicit resolution method has the advantage of preserving the form of mathematical formulae used in physical models. For example, the conservation of momentum $m \times a = \sum F_{External}$ for a given fragment translates to

```
equation m*a=sigma(quantity f:
  External_Force(f,self)|f);
```

[4] If $s_i = -$ we can always substitute $x_i$ with $x_i'$ where $x_i' = -x_i$.

where `External_Force` is the binding predicate.

Explicit resolution enables a fragment to screen influences. This complies with the concepts of encapsulation and protection of objects. However, it might be in slight contradiction with the notion of modeling from *first principles*. The implicit influence resolution of QP theory warrants that an object cannot escape the action of physical processes (like gravity). It will be up to the modeler to make sure that its model fragments are properly influenced.

### Self-Instantiation

Physical processes of QP theory are first class entities. They have two specific characteristics. First, only processes can entail direct influences. Second, their instantiation is controlled by a rule that detects *individuals* (participating objects) and fires if certain *structural* constraints are satisfied.

The language implements processes as *self-instantiating* model fragments. Self-instantiation is defined by an `instantiate` constraint. It is similar to a `rule` because it contains a context and a trigger. The context defines the set of individuals. The trigger expresses the structural conditions.

The activity of the individuals and the trigger form a *logical support* which will sustain the activity of the process. The support is compromised if an individual becomes inactive or if the trigger fails to be satisfied.

As we mentioned earlier, inactive fragment instances are not destroyed. Therefore, an inactive self-instantiated fragment will regain activity if its logical support is restored. This also means that a process is instantiated only once for a given set of individuals.

The formal definition of `instantiate` follows: given the model fragment `A`, given the global variables $x_1, \ldots, x_n$ the context $T_1\ y_1, \ldots, T_m\ y_m$ where $y_1, \ldots, y_m$ are the local variables of respective type $T_1, \ldots, T_m$, given the boolean expression
$trig(x_1, \ldots, x_n, y_1, \ldots, y_m)$;
the constraint declared within `A`
`instantiate` $T_1\ y_1, \ldots, T_m\ y_m$:
$trig(x_1, \ldots, x_n, y_1, \ldots, y_m)$;
is logically equivalent to $\forall y_1 \in T_1, \ldots, y_m \in T_m$ :
$trig(x_1, \ldots, x_n, y_1, \ldots, y_m) \Rightarrow A(y_1, \ldots, y_m)$ where $A(y_1, \ldots, y_m)$ is the fact corresponding to the instance of `A` created with the context $T_1\ y_1, \ldots, T_m\ y_m$.

As a reminder, fragment instances are equivalent to the fact representing them in the KB. Therefore, the logical expression implies the existence of the instance itself. Furthermore, existence is a synonym for activity for a user defined fragment. Therefore, the expression also implies activity.

If a self-instantiating fragment contains a `create` function, then this constructor will be applied for each instantiation.

```
fragment AgingProcess{
  instantiate WineBottle b, Pyramid p: In(b,p);

  quantity aging, taper;
  derive b.age, aging;

  state start{};
  state fast{
   derive aging, taper;
  };

  state normal{
    equation aging=1.0;
  };

  transition(start,fast,true)
  {
    aging := 10.0;
  };

  transition(fast,normal,aging<=1.0);

  fun AgingProcess create()
  {
    Assert(IsConst(taper));
    taper := -10.0^-5.0;
  }
};
```

Figure 1: `AgingProcess` model fragment.

As mentioned earlier, a fragment instance with states is set to `start` for each activation. So intermittent processes are instantiated once and they are always restored in their `start` state (if any).

Example: some people believe that wine ages faster in pyramids. Let us assume that the aging effect tapers off with time. Here is a model of this aging process. Let `p` be an instance of the `Pyramid` fragment. Let `WineBottle` be a fragment with the quantity `age`. Let `In(b,p)` be the predicate indicating that the bottle `b` is in the pyramid `p`. The `AgingProcess` fragment is described in figure 1. Let us assume that the `WineBottle` instance `b` is in `p`. We have `In(b,p)`. This fact instantiates `AgingProcess(b,p)`.

At the beginning, `b` ages ten times as fast. This rate is set by the immediate transition from `start` to `fast`. The effect tapers off in approximatively 10 days. When `aging` reaches 1 the transition from `fast` to `normal` is performed. The process remains in state `normal` until `In(b,p)` is retracted.

If we take the bottle out of the pyramid and put

it back (i.e.: retract and re-assert `In(b,p)`) the aging process restarts. Adding a second bottle `c` in `p` will entail the process `AgingProcess(c,p)`.

Notice that we could assert `In(b,p')`, where `p'` is another pyramid. This would trigger a runtime error because `b.age` would be directly influenced twice by two `derive` constraints. So, if we decide to build pyramids within pyramids to age the wine even faster we will have to use more complex models.

## Simulator

Our current implementation of the simulator is an interpretor very similar to Pika (Franz G. Amador & Weld 1993). Unlike Pika it is coded in C++ as a self-contained module. Its main components are:

**Interpretor** parses the fragment models and executes the procedural parts of the simulation.

**Knowledge Base (KB)** stores and retrieves the facts of the simulation.

**Pattern-Directed Inference Engine** applies rules to the KB.

**Logic Based Truth Maintenance System (LTMS)** records the inferences of the engine and applies the clauses (Forbus & de Kleer 1993).

**Solver** maintains and solves symbolically the algebraic equation systems.

**Integrator** maintains the list of integral variables and integrates them.

An LTMS is not a *complete* inference system. There are facts that it cannot prove nor refute. It is complete only if we use Horn Clauses (Forbus & de Kleer 1993). The solver uses dependency between variables and equations to solve unknowns. All the solved systems are cached in case they are reactivated later. The integrator uses the forward Euler method.

The inference engine detects first time instantiation and transition triggering. The simulator relies heavily on the LTMS to assemble the constraint environment. This includes transition re-triggering, `sigma` expansions, reactivation of equations, cached systems and instances.

The simulation algorithm is described in figure 2.

## Case Study

We tested the modeling language and its simulator by implementing web-based maintenance simulations. There were two scenarios featuring simple hydraulic systems and a maintenance task to accomplish. We used a client-server architecture. The client is a Java applet displaying a schematic of the hydraulic system and a control panel (see Fig3&4). The server is our simulator wrapped in a socket interface. The user issues maintenance actions by clicking the interface. The orders are sent to the simulator. The server sends periodically the state of the system to update the interface. We developed an ontology for the hydraulic domain. It uses second order differential equations and supports pressurized systems. Because of their semi-qualitative nature, our models are substantially more complex than in (Collins & Forbus 1989). The piping structure was also more detailed. We added couplings and ports to simulate assembly tasks. Special model fragments were used to detect hazardous processes such as leaks and decompressions. The simulator ran in real-time except when new equation systems were being created or processes instantiated. The current switch time is between one and two seconds for the second test case and negligible for the first.

```
load scenario and fragments
while(run)
 while(pending transitions or instantiations)
  perform transition or instantiation
 end while
 update current equation system
 solve system
 if no transitions or instantiations pending then
  update list of integral variables
  integrate
 end if
end while
```

Figure 2: Simulation algorithm.

## Analysis and Future Work

The modeling language proved practical for the development of a semi-qualitative ontology. The main difficulty was designing the models, not coding them.

The language needs some improvement. The status representation of quantities (unknown, integral, and constant) can be improved by using constraints (as in (Kuipers 1994)). Rules could be extended to entail constraints. Inheritance has to be added. Finally, adding hierarchical finite state machines will make modeling complex devices or agents easier.

## Related Work

More "qualitative" simulations are generated by the semi-qualitative simulators Q2 (Kuipers 1994) and Q3 (Berleant & Kuipers 1998). They use interval algebra. Q3 can iteratively converge to a numeric solution by iteratively refining the width of its intervals.

SIMGEN Mk3 (Forbus & Falkenhainer 1995) is used to produce web-based self-explanatory simulations

(QRG ). Similarly, CyclePad provides distributed tutoring in the domain of thermodynamics (Forbus *et al.* 1998).

Hybrid bond graphs are an alternative formalism for modeling physical systems in terms of flows of matter and energy. They are used in the HyBrSim (Mosterman & Biswas 99) simulation environment.

On the side of object-oriented languages, R++ (D.Dvorak 1995) is C++ with rule constructs.

## Conclusion

We presented the early developments of a new semi-qualitative modeling language. Our aim is to merge the concepts of compositional modeling and qualitative physics into a traditional object-oriented framework. The result give a modeling environment featuring various computational primitives such as functions, rules, finite state machines and linear differential algebraic systems. The language resolves influences in an explicit form, similar to traditional physics formulations.

Our case study demonstrated the feasibility of the simulator. It also proved that the language could tackle one of qualitative reasoning favorite domain (i.e.: hydraulic systems).

## Acknowledgment

This research is partially supported by the U.S. Air Force through Delivery Order # 17 on F41624-97-D-5002.

We want to thank Hogeun Shin for developing the web-based architecture of the case study, as well as the anonymous reviewers whose comments helped improving this paper.

## References

Aho, A.; Sethi, R.; and Ullman, J. 1985. *Compilers : Principles, Techniques, and Tools.* Addison-Wesley.

Alur, R.; Courcoubetis, C.; Halbwachs, N.; Henzinger, T. A.; Ho, P.-H.; Nicollin, X.; Olivero, A.; Sifakis, J.; and Yovine, S. 1994. The algorithmic analysis of hybrid systems. Technical Report TR94-1451, Cornell University, Computer Science Department.

Badler, N., and Erignac, C. 1998. Logistic research technilogy, product modeling technology for automating maintenance instructions, final report. Technical report, CIS, University of Pennsylvania, Philadelphia, PA.

Berleant, D., and Kuipers, B. 1998. Qualitative and quantitative simulation: bridging the gap. *Artificial Intelligence* 95(2):215–255.

Cellier, F. E. 1991. *Continuous System Modeling.* Springer Verlag.

Collins, J., and Forbus, K. 1989. Building qualitative models of thermodynamic processes. In *Proc. 3rd Int. Workshop on Qualitative Physics.*

D.Dvorak. 1995. *R++ User Manual.* Bell Labs.

de Kleer, J., and Brown, J. S. 1984. Qualitative physics based on confluences. *Artificial Intelligence* 24:7–83. Also in *Readings in Knowledge Representation*, Brachman and Levesque, editors, Morgan Kaufmann, 1985, pages 88-126.

Deshpande, A.; Göllü, A.; and Semenzato, L. Shift programming language and run-time system for dynamic networks of hybrid automata. Technical report, Univ. of Calif. at Berkeley, EECS Dept.

EAI/Transom. http://www.transom.com.

Falkenhainer, B., and Forbus, K. D. 1991. Compositional modeling: Finding the right model for the job. *Artificial Intelligence* 51(1-3):95–144.

Falkenhainer, B.; Farquhar, A.; Bobrow, D.; Fikes, R.; Forbus, K.; Gruber, T.; Iwasaki, Y.; and Kuipers, B. 1994. Cml: A compositional modeling language. Technical Report KSL-94-16, Knowledge Systems Laboratory, Stanford.

Forbus, K. D., and de Kleer, J. 1993. *Building Problem Solvers.* Cambridge, MA: MIT Press.

Forbus, K. D., and Falkenhainer, B. 1990a. Self-explanatory simulations: An integration of qualitative and quantitative knowledge. In *Proc. 8th National Conf. on Artificial Intelligence (AAAI-90)*, 380–387. AAAI Press/The MIT Press.

Forbus, K. D., and Falkenhainer, B. 1990b. Self-explanatory simulations: Scaling up to large models. In *Proc. 10th National Conf. on Artificial Intelligence (AAAI-92)*, 380–387. AAAI Press/The MIT Press.

Forbus, K. D., and Falkenhainer, B. 1995. Scaling up self-explanatory simulators: Polynomial-time compilation. In *IJCAI95.*

Forbus, K.; Everett, J.; Ureel, L.; Brokowski, M.; Baher, J.; and Kuehne, S. 1998. Distributed coaching for an intelligent learning environment. In *Proceedings of QR98.*

Forbus, K. D. 1984a. Qualitative process theory. *Artificial Intelligence* 24:85–168.

Forbus, K. D. 1984b. Qualitative process theory. Technical Report Technical Report 789, MIT AI Laboratory.

Franz G. Amador, A. F., and Weld, D. S. 1993. Real-time self-explanatory simulation. In *Proc. 11th National Conf. on Artificial Intelligence (AAAI-93).* AAAI Press/The MIT Press.

Iwasaki, Y.; Farquhar, A.; Fikes, R.; and Rice, J. 1997. A web-based compositional modeling system for

sharing of physical knowledge. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 494–500. San Francisco: Morgan Kaufmann Publishers.

Johnson, L., and Rickel, J. 1997. Steve: An animated pedagogical agent for procedural training in virtual environments. *SIGART Buletin* 16–21.

Kuipers, B. J. 1994. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. Cambridge, MA: MIT Press.

Meyer, B. 1991. *Eiffel: The Language*. Prentice Hall.

Mosterman, P. J., and Biswas, G. 1997. Formal specifications for hybrid dynamical systems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, 568–577. San Francisco: Morgan Kaufmann Publishers.

Mosterman, J., and Biswas, G. 99. A java implementation of an environment for hybrid modeling and simulation of physical systems. In *ICBGM99*.

QRG.
http://www.qrg.ils.nwu.edu/ideas/sesidea.htm.

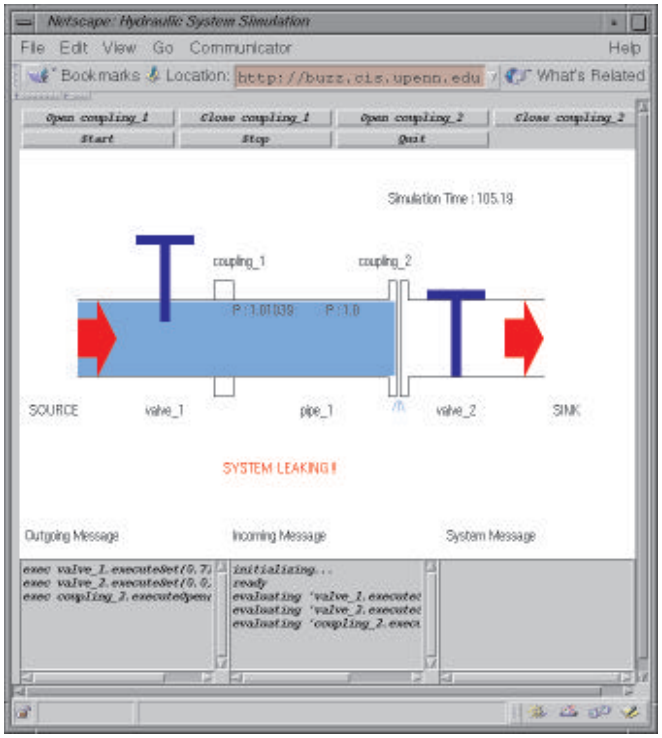


Figure 3: The first test case features a pipe (`pipe_1`) between two valves (`valve_1&2`). The assembly is placed between a water source and sink. The maintenance task is to disconnect the pipe from the circuit without creating leaks nor decompressions. The user can control the valves and disconnect the pipe by opening the couplings. The snapshot shows the pipe disconnected from the second valve and leaking water in the environment.

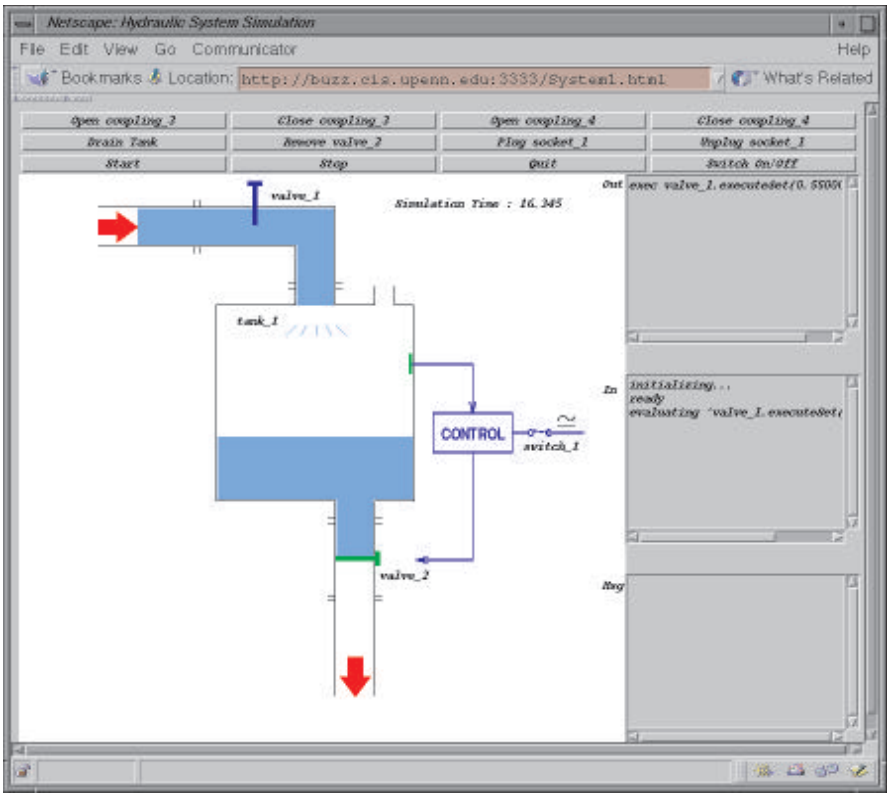


Figure 4: The second test case is a tank whose water level is regulated. The goal of the maintenance task is to remove the servo-valve (`valve_2`) without leaks or risk of electrocution. High-level actions, *Drain Tank* and *Remove Valve_2*, perform the task automatically. They are modeled as hierarchical plans and executed by autonomous processes.