System specification

COMP-522 Fall Term 2002

Hans Vangheluwe

When studying existing systems, *observations* (of structure and behaviour) are the only tangible artifacts we have at our disposal [Kli85]. A modeller may, based on observations and/or insight, build progressively more complex models of a system. Here, we present a hierarchy of abstract model structures. Each structure elaborates on the previous one, introducing (and representing) more detailed knowledge about the system. The reverse operation, going from a model containing more information to a less detailed one, must be shown to be possible. This, as some questions about the behaviour and structure of the system are better answered at lower levels in the hierarchy. In particular, explicit behaviour in the form of input and output trajectories, described at the lowest level, is often required.

In object-oriented terminology, a simulation *model* consists of model *objects* (often used to represent real-world objects, entities, or concepts) as well as *relationships* among those objects. In general, a model object is anything that can be characterized by one or more *attributes* to which *values* are assigned. Attributes are either called *indicative* if they describe an aspect inherent to the object or *relational* if they relate the object to one or more other objects. The values assigned to attributes have a *type* in the programming language sense (*i.e.,* , a set of allowed values they must belong to).

Mathematical *sets* and operations defined on those sets are the starting point for abstract system representation or modelling. Simple finite sets of numbers $\{1,2,\ldots,9\}$, identifiers $\{a,b,\ldots,z\}$, as well as infinite sets such as $\mathbb{N},\mathbb{N}^+,\mathbb{R}$, and $\mathbb{R}^+$ are typically used. Often, specific *meaning* is given to sets and their members. The set EV for example is a finite set denoting arrival and departure *events* in a queueing system

$$EV = \{ARRIVAL, DEPARTURE\}.$$

As in the *discrete event* abstraction, discussed later in greater detail, only a finite number of events are assumed to occur in a bounded time interval, the non-event symbol ϕ is introduced to denote the absence of events changing the state of the system. The event set is subsequently enriched with ϕ

$$EV^{\phi} = EV \cup \{\phi\}.$$

This demonstrates the use of basic set operations such as $\cup$. To describe multiple attributes of a system, the set product $\times$ is used

$$A \times B = \{(a,b) | a \in A, b \in B\}.$$

1 Time base and Segments

Every simulation model must have an *indexing attribute* which, at some level of abstraction will enable state transitions [Nan81]. *Time* is the most common indexing attribute. Time is special in that it inexorably progresses: the current state and behaviour of a system can only modify its future, never its past. This concept is often called *causality*: a cause must always occur before a consequence. In a simulation context, the indexing attribute is referred to as *system time*. Any set T can serve as a formalisation of time. A *nominal* relationship $=$ may be added to T to denote equality. To obtain a usable *time base* however, an *order* relation on the elements of T is needed:

$$TimeBase = \langle T, < \rangle$$

This relation has properties

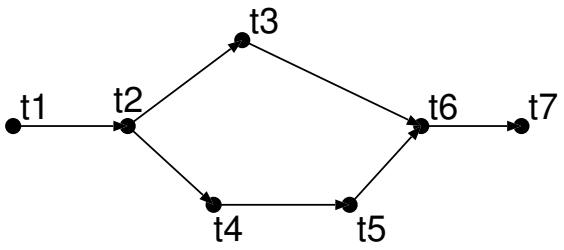

Figure 1: Partially ordered time base

- transitive: $A < B \wedge B < C \Rightarrow A < C$,
- irreflexive: $A \not< A$,
- antisymmetric: $A < B \Rightarrow B \not< A$.

This formalises the notion of order in time. The ordering relationship may be *total* (linear): each element of T can be related to every other element. A *partial* ordering where not all elements of T can be compared is useful in modelling uncertainty or concurrency. In Figure 1 for example, the nodes denote time instants and the edges denote "precedes in time" ($<$). t_2 precedes both t_3 and t_4 in time, but no information is available about the relative position in time of t_3 and t_4. In case of concurrent behaviour, causality must not be violated within the individual concurrent threads, but the time-ordering between concurrent events may be left unspecified [Mil93]. Mathematically, partial ordering leads to a lattice structure.

For total ordering, it must be possible to compare any two elements of T:

$$\forall t, t' \in T : t < t' \vee t' < t \vee t = t'.$$

In case of total ordering, *intervals* may be defined. With intervals, the past $T_{t[}$ and future $T_{]t}$ of an instant $t \in T$ may be defined

$$T_{t[} = \{\tau | \tau \in T, \tau < t\},$$
$$T_{]t} = \{\tau | \tau \in T, t < \tau\}.$$

Once intervals have been defined,

$$T_{\langle t_b, t_e \rangle}$$

denotes a time interval, where $\langle t$ means $]t$ or $[t$. In many cases, $(T, +)$ is an Abelian group with zero 0 and inverse $-t$. In case $+$ is order preserving

$$t_1 < t_2 \Rightarrow t_1 + t < t_2 + t.$$

Common time bases (with appropriate $<$ and $+$) are

- $T = \{NOW\}$. Models such as algebraic models are *instantaneous*. The time base is a singleton.
- $T = \mathbb{R}$. Models with this time base are called *continuous-time* models. Note how *discrete event models* have $\mathbb{R}$ as a time base. However, only at a finite number of time-instants in a bounded time-interval, an event different from the non-event ϕ occurs.
- $T = \mathbb{N}$ (or isomorphic). Models with this time base are called *discrete-time* models. Some formalisms such as Finite State Automata (FSA) do not have an explicit notion of time (unlike their extension, timed automata). Hence, they are often called *untimed* models. There is however a notion of progression (from one state to another). According to our general definition, the index of progression, a natural number, is time.

In *hybrid* system models which combine aspects of continuous and discrete models [MB01], a system evolves continuously over time ($\mathbb{R}$) until a certain condition is met. Then, *instantaneously* (the continuous time does not progress), the system may go through a number of discrete states (the index of progression is discrete) before continuing its continuous behaviour. To uniquely describe progression (of generalized time) in this case, a tuple (t_c, t_d) depicted in Figure 2 is

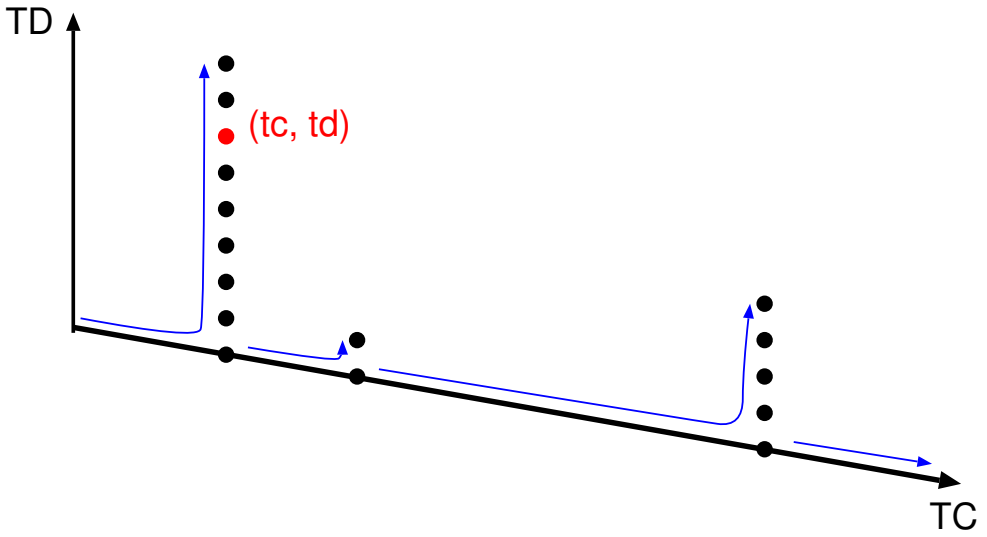

Figure 2: Time base for hybrid system models

needed. Even when a series of discrete transitions keeps returning to the *same* state, the discrete index t_c allows one to distinguish between them. The time base used is

$$T = \{(t_c, t_d) | t_c \in \mathbb{R}, t_d \in \{1, \ldots, N(t_c)\}\}.$$

Here, $N(t_c)$ (≥ 1) describes the number of discrete transitions the system goes through at continuous time t_c. Obviously, only a partial ordering will be defined over T which consists of first testing the relationship between the t_c components, and subsequently (if equal), that between the t_d components.

In case of Partial Differential Equations (PDEs), the time base remains $\mathbb{R}$. The other *independent variables* (often space in the form of some coordinate system) should be seen as infinitely many state-variable *labels* or *generalized coordinates*.

1.1 Behaviour

Given a time base, we wish to formalize *behaviour* over time. This is done by means of a time function, called *trajectory* or *signal*

$$f : T \to A$$

describing, at each time t, the value of the signal. A denotes the set of valid values f can take over T. The time base may be restricted to a subset of T: $T' \subseteq T$. The restriction of f to T' is

$$f|T' : T' \to A,$$

$$\forall t \in T' : f|T'(t) = f(t).$$

The past of f is defined as $f|T_{t\rangle}$. The future of f is defined as $f|T_{\langle t}$.

The restriction of f to an interval is called a *segment* ω

$$\omega : \langle t_1, t_2 \rangle \to A.$$

The set of all allowed segments is called Ω. It is a subset of all possible segments (A, T). The length of a segment:

$$l : \Omega \to T_0^+$$

$$\omega \to t_f - t_i, dom(\omega) = \langle t_b, t_e \rangle$$

Segments are contiguous if their domains $\langle t_1, t_2 \rangle$ and $\langle t_3, t_4 \rangle$ are contiguous: $t_2 = t_3$.

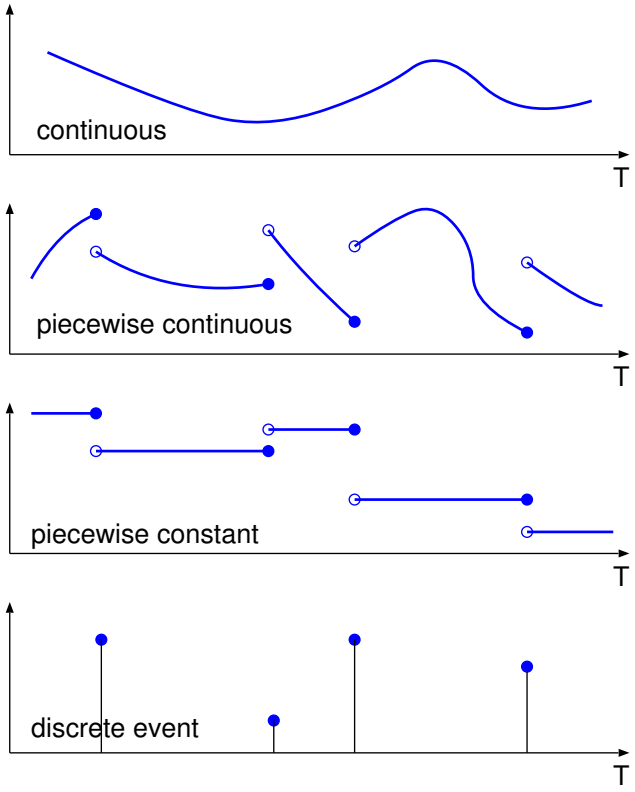

Figure 3: Segment types

Contiguous segments may be concatenated – $\omega_1 \bullet \omega_2$:

$$\omega_1 \bullet \omega_2(t) = \omega_1(t), \forall t \in dom(\omega_1);$$

$$\omega_1 \bullet \omega_2(t) = \omega_2(t), \forall t \in dom(\omega_2),$$

where $\langle$ and $\rangle$ must denote matching open/closed interval boundaries to ensure the concatenated segment is still a *function* (*i.e.,* has a unique value in each point of its domain).

A desirable property of a set of segments Ω is that it is closed under concatenation $\bullet$: concatenating *any* left and right segment of a segment yields the same segment:

$$\forall t \in dom(\omega) : \omega_{t\rangle} \bullet \omega_{\langle t} = \omega.$$

Figure 3 shows some common segment types: continuous, piecewise continuous, piecewise constant and discrete event. Note how for discrete event systems, inputs and output segments are *event segments*

$$\omega : \langle t_1, t_2 \rangle \to A \cup \{\phi\},$$

with ϕ the non-event. For such systems, the internal state behaviour is piecewise constant (the internal state only changes at event times).

2 Levels of system specification

With a time base and segments defined, we can build a hierarchy of system specification structures which incorporate progressively more knowledge about the system. All these structures will view the system as a *box* interacting with its environment through a well defined *interface*. The levels presented here elaborate on the hierarchy first proposed by Klir [Kli85] and later modified by Zeigler [ZPK00].

2.1 Observation Frame

At the lowest level, the only knowledge we have of the behaviour of a system is how we wish to observe it: which time base to use and which quantities to observe at instants from the time base. This is represented in the form of an Observation Frame O:

$$O \equiv \langle T, X, Y \rangle.$$

T with appropriate operators forms a time base. X is the input value set. It is a model for the input (influencing the behaviour of the system) variables we consider. Y is the output value set. It is a model for the system response variables.

2.2 I/O Relation Observation

Once the interface variables to observe as well as their value ranges have been determined, all possible *relationships* between input and output segments can be recorded

$$IORO \equiv \langle T, X, \Omega, Y, R \rangle.$$

Here, $\langle T, X, Y \rangle$ is an Observation Frame, and Ω is the set of all possible input segments for this system. Note how Ω allows one to specify how the system's *environment* may influence the system. As such, Ω formalizes the Experimental Frame's generator presented before. Ω is a subset of all mathematically possible segments with T as domain and X as image. R is the *I/O relation*

$$R \subseteq \Omega \times (Y, T),$$

where $\Omega \subseteq (X, T)$, all possible segments with T as domain and X as image. (Y, T) stands for all possible segments with T as domain and Y as image. Input segments ω and output segments ρ are defined as

$$\omega : \langle t_i, t_f \rangle \rightarrow X;$$

$$\rho : \langle t_i, t_f \rangle \rightarrow Y.$$

Though not necessary, it is common to observe input and output segments over the same time domain. The relation R relates input and output segments

$$(\omega, \rho) \in R \Rightarrow dom(\omega) = dom(\rho).$$

As will be discussed further on, general *non-causal* relationships between interface variables, not specifying *a priori* which are input and which are output may be specified by R. Higher levels in the specification hierarchy are explicitly causal.

2.2.1 From I/O Relation Observation to Observation Frame

It is possible to go from an I/O Relation Observation model specification to an Observation Frame level model by merely discarding the Ω and R information at the I/O Relation Observation level.

2.3 I/O Function Observation

At the I/O Relation Observation level, an input segment ω is not necessarily associated with a unique output segment ρ. This is due to a limited knowledge of the internal working of the system. At the I/O Function Observation level, we want to associate a *unique* output segment with every input segment. Therefore, more information needs to be specified about the system. This is done in the form of a set F of *I/O functions* f. This leads to the I/O Function Observation structure

$$IOFO \equiv \langle T, X, \Omega, Y, F \rangle,$$

where $\langle T, X, Y \rangle$ is I/O Relation Observation, Ω is the set of all possible input segments, F is the set of I/O functions:

$$f \in F \Rightarrow f \subset \Omega \times (Y, T);$$

$$dom(f(\omega)) = dom(\omega).$$

f is conceptually equivalent to the system's *initial state*: For each f, an input segment will be transformed into a unique output segment.

2.3.1 From I/O Function Observation to I/O Relation Observation

It is possible to go from an I/O Function Observation to an I/O Relation Observation by constructing R from F:

$$R = \bigcup_{f \in F} f.$$

2.4 I/O System

In some cases, we have some insight into the *internal* working of the system. This insight usually consists of a number of *descriptive variables* and how their values evolve over time. Under certain conditions, these variables are *state variables*. One usually tries to keep the set of state variables as small as possible.

In general systems theory [Wym67], a causal (output is the consequence of given inputs), deterministic (a known input will lead to a unique output) system model SYS is defined. It is a template for a plethora of different formalism such as Ordinary Differential Equations, Finite State Automata, Difference Equations, Petri Nets*, etc.* Its general form is

$$SYS \equiv \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle$$

T	time base
X	input set
$\Omega = \{\omega : T \to X\}$	input segment set
Q	state set
$\delta : \Omega \times Q \to Q$	transition function
Y	output set
$\lambda : Q \to Y$ (or $Q \times X \to Y$)	output function

$$\forall \omega, \omega' \in \Omega, \delta(\omega \bullet \omega', q_i) = \delta(\omega', \delta(\omega, q_i)).$$

The time base T is the formalisation of the independent variable time. The input set X describes all possible allowed input values (possibly a product set). An input segment ω represents input during a time-interval. The history of system behaviour is condensed into a *state* (from a state set Q). The dynamics is described in a transition function δ which takes a current state, and applies an input segment $\omega \in \Omega$ to it to obtain a new state. The system may generate output. This output is obtained as a function λ of the state (and more generally, of the current input too). State and transition function must obey the composition or semigroup property as shown in Figure 4. This property, whereby a transition over a time interval $[t_i, t_f]$ can always be split into a composition of transitions over arbitrary sub-intervals, is the basis of all model simulators. Obviously, this also requires Ω to be closed under concatenation as well as left segmentation. Closure under concatenation requires that the concatenation of any two contiguous segments in Ω is again in Ω. Left segmentation requires that any left segment of a segment in Ω is an element of Ω in its own right.

As the output function is described separately, efficient simulators will *only* invoke this function (which may be large and compute-intensive) when the user needs to observe output. Note how the *output intervals* (times between outputs) are not part of the model, but rather of the simulation experiment. Figure 5 shows how output need not be produced at each transition time. Even though a model written by a user may not distinguish between δ and λ, a simple *dependency analysis* will identify which variables and expressions are not needed to compute δ. Such variables are output variables $\in Y$ and the expressions belong in λ.

As SYS is a template for a host of causal, deterministic formalisms, it is possible to describe both models of the vessel example presented earlier. In the Ordinary Differential Equation (ODE) case, the time base is continuous ($\mathbb{R}$). The transition function is written in integral form. Different numerical approximations of the integral can be used in the implementation of an abstract simulator.

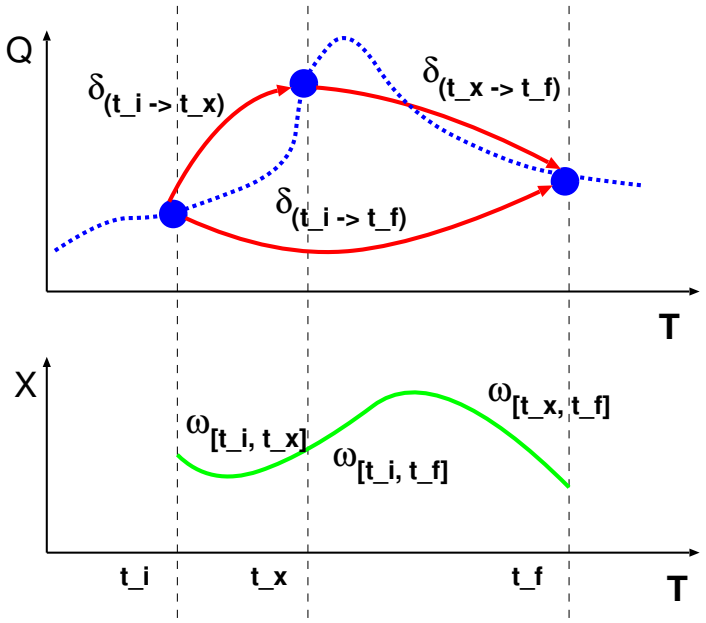

Figure 4: SYS state transition property

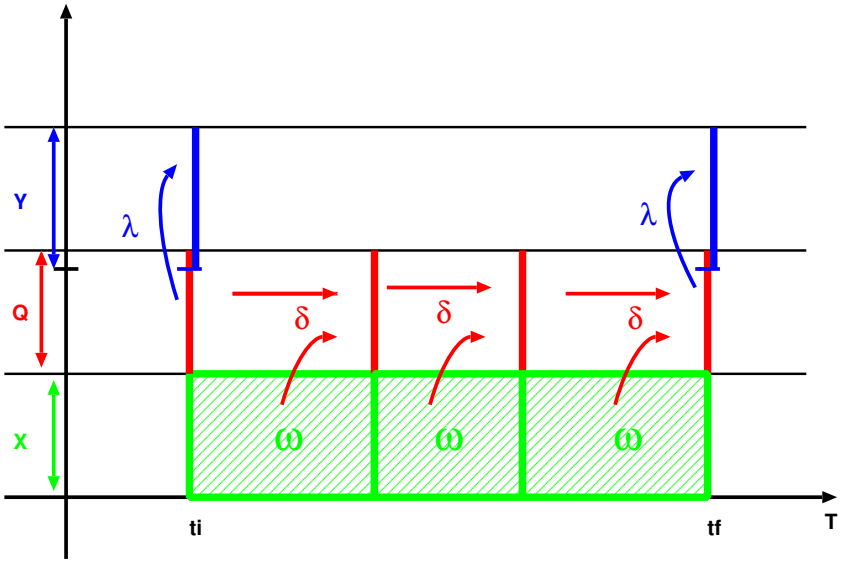

Figure 5: Simulation kernel operation

Inputs (discontinuous → hybrid model):

- Emptying, filling flow rate ϕ
- Rate of adding/removing heat W

Parameters:

- Cross-section surface of vessel A
- Specific heat of liquid c
- Density of liquid ρ

State variables:

- Temperature T
- Level of liquid l

Outputs (sensors):

- $is\_low, is\_high, is\_cold, is\_hot$

$$\begin{cases} \frac{d\mathsf{T}}{dt} &= \frac{1}{l}[\frac{W}{c\rho A} - \phi\mathsf{T}] \\ \frac{dl}{dt} &= \phi \\ is\_low &= (l < l_{low}) \\ is\_high &= (l > l_{high}) \\ is\_cold &= (\mathsf{T} < \mathsf{T}_{cold}) \\ is\_hot &= (\mathsf{T} > \mathsf{T}_{hot}) \end{cases}$$

The model can be represented in the I/O System specification form

$$SYS^{ODE}_{VESSEL} = \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$$\begin{aligned}
&T = \mathbb{R} \\
&X = \mathbb{R} \times \mathbb{R} = \{(W, \phi)\} \\
&\omega : T \to X \\
&Q = \{(\mathsf{T}, l) | \mathsf{T} \in \mathbb{R}^+, l \in \mathbb{R}^+\} \\
&\delta : \Omega \times Q \to Q \\
&\delta(\omega_{[t_i, t_f[}, (\mathsf{T}(t_i), l(t_i))) = \\
&\quad (\mathsf{T}(t_i) + \int_{t_i}^{t_f} \frac{1}{l(\alpha)}[\frac{W(\alpha)}{c\rho A} - \phi(\alpha)\mathsf{T}(\alpha)]d\alpha,\ l(t_i) + \int_{t_i}^{t_f} \phi(\alpha)d\alpha) \\
&Y = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} = \{(is_low, is_high, is_cold, is_hot)\} \\
&\lambda : Q \to Y \\
&\lambda(\mathsf{T}, l) = ((l < l_{low}), (l > l_{high}), (\mathsf{T} < \mathsf{T}_{cold}), (\mathsf{T} > \mathsf{T}_{hot})).
\end{aligned}$$

At a higher level of abstraction, we have represented time as a discrete integer index. The transition function lists all possible state transitions.

$$SYS^{DISCR}_{VESSEL} = \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle$$

$$\begin{aligned}
&T = \mathbb{N} \\
&X = \{heat, cool, off\} \times \{fill, empty, closed\} \\
&\omega : T \to X \\
&Q = \{(\mathsf{T}, l) | \mathsf{T} \in \{cold, \mathsf{T}_{between}, hot\}, l \in \{empty, l_{between}, full\}\} \\
&\delta : \Omega \times Q \to Q \\
&\delta(\omega_{[n, n+1[}[(off, fill), (cold, empty)) = (cold, l_{between}) \\
&\delta(\omega_{[n, n+1[}[(off, fill), (cold, l_{between})) = (cold, full)) \\
&\delta(\omega_{[n, n+1[}[(off, fill), (cold, full))) = (cold, full)) \\
&\qquad\qquad \vdots \\
&\delta(\omega_{[n, n+1[}[(heat, fill), (hot, full))) = (hot, full)) \\
&Y = \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \\
&\lambda : Q \to Y \\
&\lambda(\mathsf{T}, l) = ((l = low), (l = high), (\mathsf{T} = cold), (\mathsf{T} = hot))
\end{aligned}$$

The Finite State Automaton formalism [CL89]

$$FSA \equiv \langle \Sigma, S, s_0, d, F \rangle,$$

where

- Σ is the input alphabet (a finite and nonempty set of symbols),
- S is the finite nonempty set of states,
- s_0 is the initial (or start) state, $s_0 \in S$,
- $d : S \times \Sigma \to S$ is the state transition function,
- $F \subseteq S$ is the set of final or accepting states,

fits the general SYS structure presented above.

The formalism is specified by elaboration of the elements of the SYS 7-tuple:

$$SYS \equiv \langle T, X, \Omega, Q, \delta, Y, \lambda \rangle.$$

In the SYS specification, the initial state is not explicitly represented. Also, accepting states are not defined. Hence, SYS describes a parametrized *class* of Finite State Automata.

The time base

$$T = \mathbb{N} \text{ (or isomorphic with } \mathbb{N}).$$

This can be interpreted as (implicit) discrete time-clicks. It is possible to extend the FSA formalism to assign with each state and/or with each transition, a known duration.

The input set

$$X = \Sigma.$$

The set Ω of all input segments ω. An input segment encodes a *sequence* of inputs from X.

The finite state set

$$Q = A$$

enumerates all states in the automaton.

The state transition function δ transforms a current state, through input and time-advance, to a new state

$$\delta : \Omega \times Q \to Q.$$

It is obtained by iteratively applying all FSA state transitions f in an input segment ω.

The output function λ takes the form

$$\lambda : Q \to Y$$

in case of a Moore machine (the input can only influence the output *via* the state), or

$$\lambda : \Omega \times Q \to Y$$

in case of a Mealy machine (the input can *directly* influence the output).

2.4.1 From I/O System specification to I/O Function Observation

It is possible to go from an I/O System specification to an I/O Function Observation. For a given initial condition q and a given input segment ω, we can define a *state trajectory* $STRAJ_{q,\omega}$ from SYS

$$STRAJ_{q,\omega} : dom(\omega) \to Q,$$

with

$$STRAJ_{q,\omega}(t) = \delta(\omega_{t\rangle}, \forall t \in dom(\omega).$$

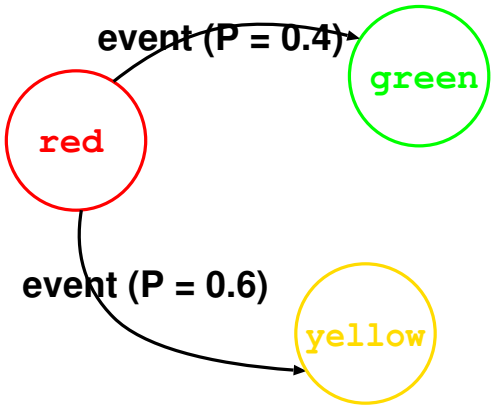

Figure 6: Non-deterministic model with transition probabilities

From this state trajectory, an *output trajectory* $OTRAJ_{q,\omega}$ may be constructed

$$OTRAJ_{q,\omega} : dom(\omega) \rightarrow Y,$$

with

$$OTRAJ_{q,\omega}(t) = \lambda(STRAJ_{q,\omega}(t), \omega(t)), \forall t \in dom(\omega).$$

Thus, for every q (initial state), it is possible to construct

$$\mathcal{T}_q : \Omega \rightarrow (Y, T),$$

where

$$\mathcal{T}_q(\omega) = OTRAJ_{q,\omega}, \forall \omega \in \Omega.$$

The I/O Function Observation associated with SYS is then

$$IOFO = \langle T, X, \Omega, Y, \{\mathcal{T}_q(\omega) | q \in Q\}\rangle.$$

Subsequently, we may derive the I/O Relation Observation by constructing the relation R as the union of all I/O functions:

$$R = \{(\omega, \rho) | \omega \in \Omega, \rho = OTRAJ_{q,\omega}, q \in Q\}.$$

In SYS, δ is *deterministic*: applying the same input segment to the same state will always lead to the *same, unique* new state (and output). Often, deterministic simulation kernels are used to simulate non-deterministic models. Two main approaches are possible:

1. A deterministic model is decorated with transition probabilities as shown in Figure 6. The same model is then simulated a number of times, with the same initial conditions and parameters. Whenever a non-deterministic transition is encountered however, a unique, deterministic, transition is chosen by sampling from a stochastic distribution, taking into account the transition probabilities in the model. Thus, from the point of view of the simulation engine, it is simulating a deterministic model. To be able to make meaningful statements about the behaviour of the non-deterministic model, a sufficient number of samples must be simulated to obtain statistically relevant *estimates* of performance metrics (such as average queue lengths in a queueing model). In discrete event simulation in particular, this approach is common and its statistical aspects have been studied in great detail [LK91]. In a slightly modified form, this approach is called Monte Carlo simulation.

2. One may wish not to specify any probability distribution but leave the uncertainty of making a transition to more than one new state in the transition function. In case of State Automata, this turns the transition graph into a transition hypergraph [Har88]. Such a specification can always be transformed into a deterministic one by constructing a new state set $Q_{new} = 2^Q$, the set of all subsets (powerset) of Q [Cas93]. A new transition function is constructed describing the –now deterministic– transition to a new state, denoting the set of states from Q to which a non-deterministic transition existed in the old model. It is noted that in quantum physics, evolution over time of a wave function (a distribution interpreted as being probabilistic) is also deterministic.

Zeigler [ZPK00] presents a refinement of *SYS* in which behaviour is specified as an *iterative* application of *generator segments*. Arbitrary input segments are generated from elementary segments.

It should be noted that though models may be iteratively simulated, this is not necessary per se. If a symbolic (analytical) solution can be found, this is often preferable. Analytical solutions usually describe a (parametrised) class of solutions rather than a single one. Also, accumulation of numerical errors is often avoided. As an example, the following model described in the Difference Equation formalisms ($T = \mathbb{N}, Q = \mathbb{R}$)

$$\begin{cases} x_1 &= 1 \\ x_{i+1} &= ax_i + 1, i > 1 \end{cases}$$

can be re-written as

$$\begin{cases} x_n &= 1 + a + a^2 + \ldots + a^{n-1} \\ ax_n &= a + a^2 + \ldots + a^{n-1} + a^n, \end{cases}$$

subtracting the second from the first equation leads to the instantaneous (no iteration required) solution

$$x_n = \frac{1 - a^n}{1 - a}.$$

3 Structured specifications

Upto now, no *structure* was explicitly specified for the sets X, Y, *etc.* at any of the above specification levels. *Orthogonal* to the specification hierarchy, at *each* of the levels, the internal structure of input, output, and state sets as well as of functions may be made explicit. This allows one to construct sets from more primitive sets.

One way of introducing structure is through *multivariable sets*. A multivariable set (one possible representation of a programming language Symbol Table) uses a finite sequence V of n distinct variable names, identifiers, labels, or references

$$V = (v_1, v_2, \ldots, v_n).$$

With each of these names will be associated a *value set* of values a variable with that name may take

$$V_1, V_2, \ldots V_n.$$

The full multivariable set is then

$$S = (V, V_1 \times V_2 \times \ldots V_n)_s.$$

A projection operator . can be defined

$$. : S \times V \to \bigcup_{j=1}^{n} V_j, S.v_i = s_i, \forall v_i \in V.$$

With A and B structured sets, a structured function may be defined

$$f : A \to B,$$

where the projection of f on a name in the image set is

$$f.b_i : A \to ((b_i), B_i)_s,$$

$$f.b_i(a) = f(a).b_i$$

In case of interfaces or ports, the names denote individual *port names*. For example, in the water pot example

$$X = ((heatFlow, liquidFlow), \mathbb{R} \times \mathbb{R})_s.$$

With $x \in X$, we may refer to the $x.heatFlow$ input port value of the model.

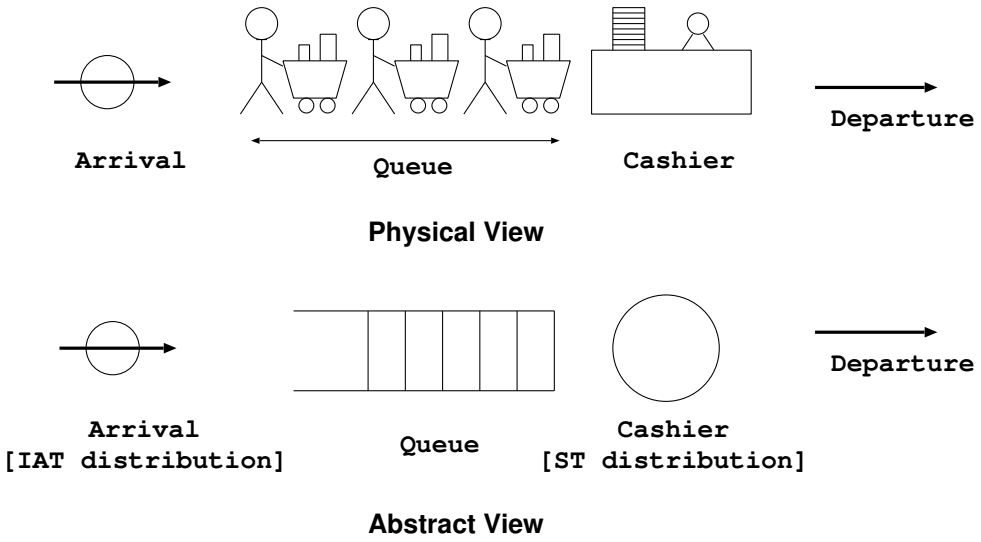

Figure 7: Simple single queue, single server system

In case of state variables, the names denote *variable names*. Again, in the water pot example

$$S = ((temperature, level),]0.0, 100.0[\times [0, H])_s .$$

With $s \in S$, we may refer to the $S.temperature$ state variable value. As such, structured sets and functions are similar to *variables and* their *types* in programming languages.

Figure 7 depicts a simple single server, single queue system. In a discrete event model of this system, the state set could be a structured set containing a simple abstraction of the queue (the queue length) and the status of the server

$$SQ = ((qLength, cashStatus), \mathbb{N} \times \{Idle, Busy\}).$$

With $s \in SQ$, we may refer to $s.qLength$.

Structured sets may be used to add structure to I/O Observation Frame, I/O Relation Observation, I/O Function Observation, as well as to I/O system formalisms.

A modelling language (such Modelica uses structured sets to allow the modeller to describe ports and variables by name (rather than by an index in a product set).

4 Multicomponent specifications

A common means to tackle complexity is to decompose a problem *top-down* into smaller sub-problems. Conversely, complex solutions may be built *bottom-up* by combining primitive sub-problem solution building blocks. Both approaches are instances of *compositional modelling*: the connection (but more general, composition) of *interacting component* models to build new models. In case the components *only* interact via their interfaces, and do not influence each other's internal working in any other way (model information is completely *encapsulated* in object-oriented terminology [Boo98, Zei97, Weg90]), the compositional modelling approach is called *modular*. If inter-components access is not restricted to take place via interfaces or ports only, the approach is called *non-modular*.

4.1 Modular multicomponent specification

A modular multicomponent specification or *coupled* model or *network* model as depicted in Figure 8 can be mathematically described as a structure

$$N = \langle T, X_{self}, Y_{self}, D, \{M_d | d \in D\}, \{I_d | d \in D \cup \{self\}\}, \{Z_d | d \in D \cup \{self\}\} \rangle .$$

In this structure,

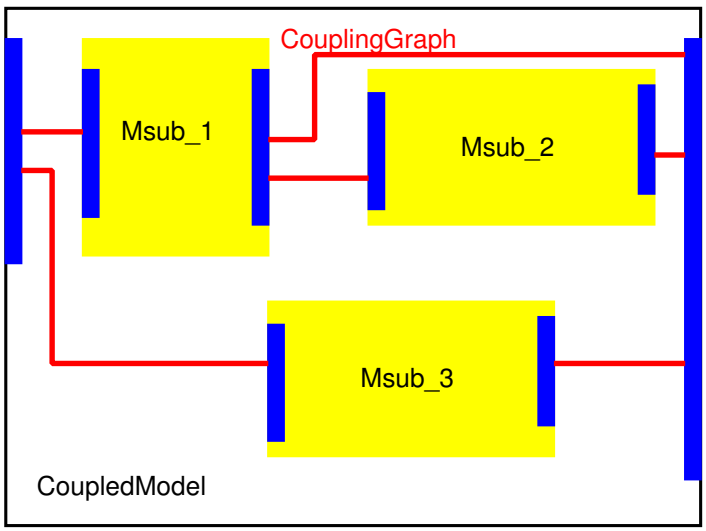

Figure 8: Example network model

- X_{self} and Y_{self} are inputs and outputs of the network N. $self$ (`this` in `C++` terminology) allows us to treat the network object itself as any other object. In a modular specification, the network will interact with its environment through X_{self} and Y_{self} *only*.
- D is a set of component *references* or *names*.
- The M_d's are component models ($\forall d \in D$).
- $I_d \subseteq D \cup \{self\}$ is the set of *influencers* of d. Alternately, the *influencees* of a component could be specified. Both allow one to specify the coupling graph topology.
- $Z_d : \times_{i \in I_d} YX_i \to XY_d$ is the *interface map* for $d \in D \cup \{self\}$

$$YX_i = X_i \text{ if } i = self,\ YX_i = Y_i \text{ if } i \neq self$$

$$XY_d = Y_d \text{ if } d = self,\ XY_d = X_d \text{ if } d \neq self$$

The interface map allows one to specify mappings or conversions required when for example an output event "departure" of one sub-model is routed to the input of another sub-model. The accepting sub-model may be expecting "arrival" events only. Thus, a "departure"–to–"arrival" mapping must be carried out to allow true sub-model re-use. Sub-models should *not* have to be modified when re-used in a network ! Together, I_d and Z_d specify the network coupling completely.

The above structure does *not* reveal anything about the overall *behaviour* of the network. It only specifies structure. As such, only questions about that structure (number of components, presence of feedback, ...) can be answered at this level. The behavioural semantics of a coupled model is often given by specifying a composition procedure, also known as *flattening*. If all the component models are specified using the same formalism, the composition procedure may replace the network by a single model in that same formalism. In that case, the formalism is said to be *closed under composition*. This is obviously a highly desirable property. It is the basis for

- implementing a simulation kernel which implements the simulation of coupled models by orchestrating simulators of the components,
- finding the meaning of *hierarchies* of models by applying the composition procedure recursively.

Compositional modelling may be done (as long as a composition procedure is given) at *any* of the specification levels mentioned before (I/O observation frame, I/O Relation Observation, I/O Function Observation, I/O System). The semantics is given by the elaboration of closure under coupling. Within the I/O System level, there are obviously many formalisms for which a composition procedure may be defined.

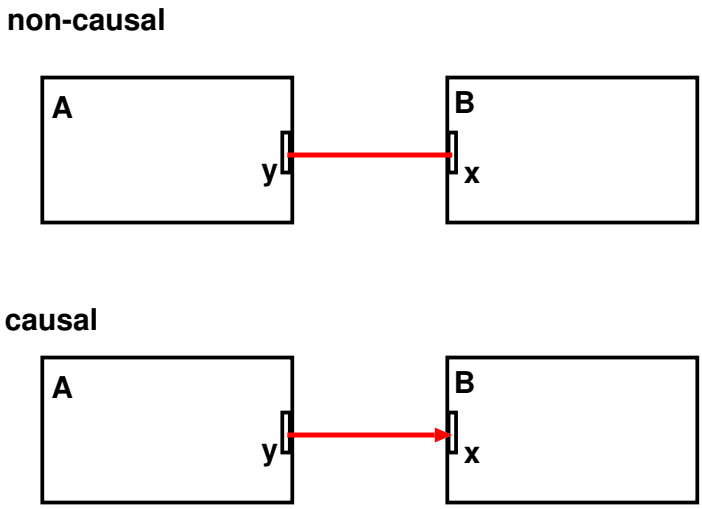

Figure 9: Composition for DAE models

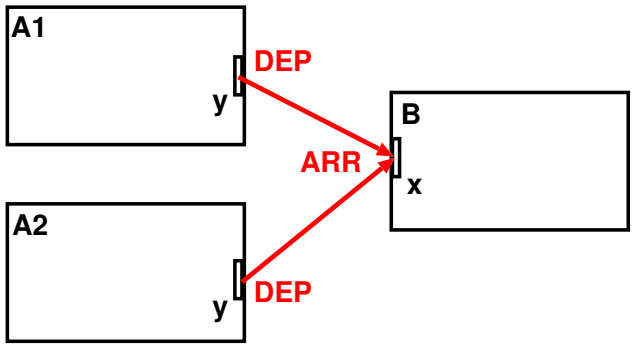

Figure 10: Composition for Discrete Event models

Assuming the component models M_d in

$$\langle T, X_{self}, Y_{self}, D, \{M_d | d \in D\}, \{I_d | d \in D \cup \{self\}\}, \{Z_d | d \in D \cup \{self\}\}\rangle$$

are all specified at the I/O System level, implementing closure would mean this model is replaced by

$$\langle T, X_{self}, \Omega, Q, \delta, Y_{self}, \lambda\rangle.$$

In continuous models, composition is achieved by replacing connections by algebraic equalities and combining these with the components' mathematical equations into one large set of equations (see Figure 9). In discrete event models, composition is achieved by describing how different component events are globally *scheduled* in a causality-preserving order (see Figure 10). Both cases will be described in more detail later.

If structured sets are used to describe input and output sets of the network as well as of the components, connections may be described between specific (named) ports making the mathematical structure more intuitive and closer to the form commonly used in modelling languages.

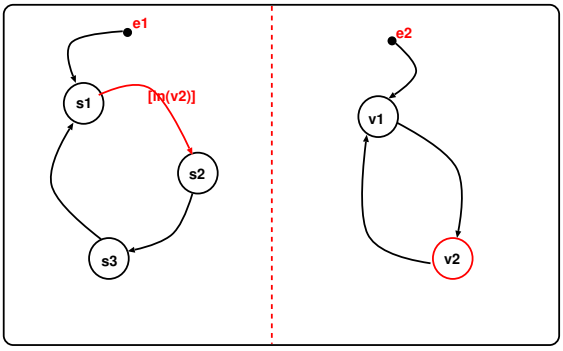

Figure 11: Composition of Statechart components

4.2 Non-modular multicomponent specification

Often, models in some formalism are graphically specified by connecting building blocks. Though this gives the illusion of being modular, often the transition functions of the building blocks interact *directly*, without going through interfaces. This limits

- the insight into the semantics a user gets directly from the graphical representation,
- the potential for component-based re-use as a component depends on its environment in unexpected ways,
- the potential for distributed implementation as hidden component interactions need to be passed as messages between distributed processors.

In programming languages, non-modularity corresponds to the use of *global variables*, and to the ensuing *side-effect* functions may have. Object oriented design discourages this in encapsulating information inside objects (instances of classes) and only giving access through well-defined interface methods.

The process interaction discrete event language GPSS [Sch74] is a notable, but popular, example of non-modular model specification. In his 1992 thesis, Claeys [Cla92] has added encapsulation and hierarchy to GPSS, making it more modular. A similar approach, modifying the process interaction world view, was advocated by Cota and Sargent [CS92].

Statecharts [Har88, HN96, HG97] are modular in their treatment of external events, but non-modular in their treatment of concurrent (orthogonal) behaviour. In Figure 11, modular coupling of the model to its environment is possible via the input-ports e_1 and e_2. The concurrent state automata (concurrency is denoted by the dotted vertical line) however interact in a non-modular fashion: a transition in the left automaton from s_2 to s_1 will occur if the automaton on the right is in state v_2.

The mathematical representation of a nonmodular multicomponent system is

$$MC = \langle T, X, \Omega, Y, D, \{M_d | d \in D\} \rangle$$

where

$$M_d = \langle Q_d, E_d, I_d, \delta_d, \lambda_d \rangle, \forall d \in D.$$

In the above,

- D is a set of component *references* or names,
- Q_d is the *state set* of component d,
- $I_d \subseteq D$ is the set of *influencers* of d,
- $E_d \subseteq D$ is the set of *influencees* of d,
- δ_d is the *state transition function* of d
  $\delta_d : \times_{i \in I_d} Q_i \times \Omega \to \times_{j \in E_d} Q_j$.
  The new states in all of the influencees are determined by the old states in all of the influencers.
- λ_d is the *output function* of d
  $\lambda_d : \times_{i \in I_d} Q_i \times \Omega \to Y$
  Output is determined by the state of all the influencers.

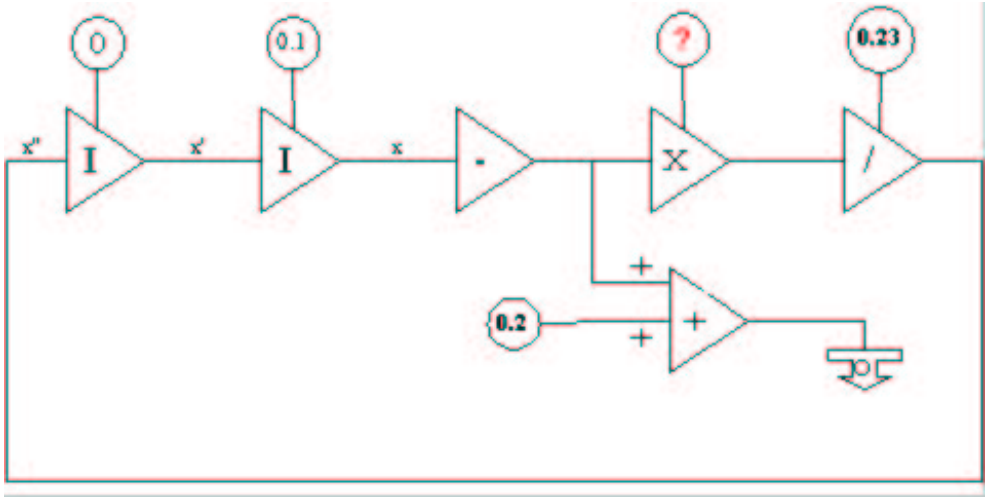

Figure 12: A causal block diagram model for the Mass-Spring problem

Together, the nonmodular multicomponent system is an I/O System level specification. Its state set is the product set of all component state sets and its transition function is given by combining all individual component transition function. Note how the components are themselves *not* I/O Systems and it is thus not possible to construct a hierarchy of nonmodular multicomponent systems. Also, the particular structure of these systems implies they only make sense at the I/O Systems level.

In Figure 12, we show the Mass-Spring model presented when discussing the modelling and simulation process, described in the *causal block diagram* formalism. In the figure, `I` blocks denote Integrators. The other blocks denote algebraic operations. Causal block diagram models can be given nonmodular multicomponent semantics with

- $E_d = \{d\}$
- $Y = \times_{d \in D} Y_d$
- $Q = \times_{d \in D} Q_d$
- $\delta(q, \omega).d = \delta_d(\times_{i \in I_d} q_i, \omega)$
- $\lambda(q, \omega(t)).d = \lambda_d(\times_{i \in I_d} q_i, \omega(t))$

5 Non-causal modelling

Though quite generic, the formalisms presented above at the I/O systems level do only describe *causal* models. When describing physical systems, *non-causal* models express conservation laws and constraints without imposing a *computational causality*. The dynamics of a simple resistor for example (Figure 13) is described by Ohm's law

$$V - Ri = 0.$$

Depending on whether the current i through (when connected to a current source) or the voltage drop V over (when connected to a voltage source) the resistor is known, the causal equations

$$V := Ri$$

or

$$i := \frac{V}{R}$$

would be used. Mathematically, those two expressions are evidently equivalent, but if we are interested in writing a *computational block*, with an input and an output, they are distinct. In the case of the resistor, which, as a physical *object*, is non-causal in nature, a choice can not be made a priori as it depends on what the resistor will be connected to. As such, it does not make sense to define two objects (as in Matlab/Simulink), one for each causal orientation. The whole problem

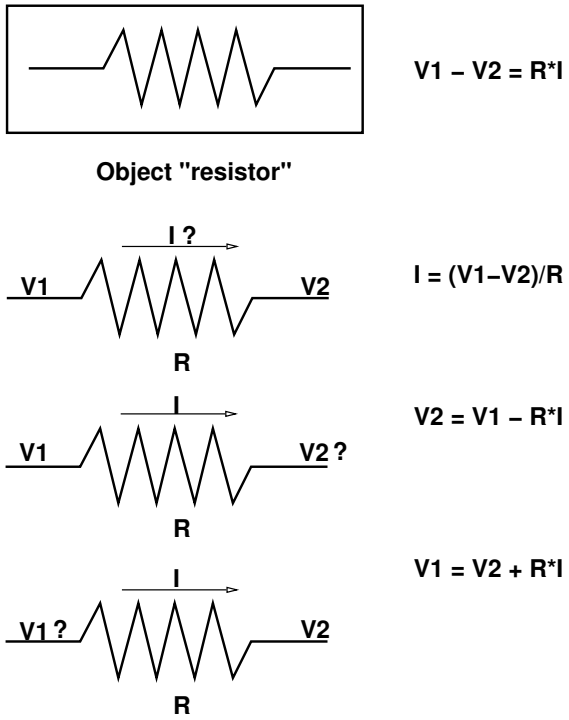

Figure 13: Electrical resistor

stems from the fact that we are dealing with assignment instructions, which we have stressed by using the assignment operator :=, and not mathematical equalities. One says that computational blocks are *causally oriented*. From the point of expressiveness and *re-usability* (in different causal contexts), the non-causal, implicit representation is preferred. From a computational point of view however, the causal representation is preferred as solving implicit equations, though possible, is highly inefficient compared to solving their causal counterparts. Inefficiency is mainly due to the iterative nature of implicit solvers. As will be demonstrated later, it is in many cases possible to automatically transform a non-causal representation into a causal one once inputs and outputs have been specified.

Most formalisms and corresponding modelling languages are causally oriented, which strongly limits the *re-usability* of sub-models, and in general, their fitness to serve as a basis for the construction of general sub-model languages. Causally non-oriented formalisms and languages (non-causal, for short) exist as well, and some of them will be presented further on.

6 Classifications

Previously, the importance of making the *process* of modelling and simulation explicit was stressed. By doing so, different sub-processes could be identified, thus simplifying the generic (application-independent) study of modelling and simulation. Now, a further classification of different types of models, their corresponding simulation kernels, and supporting methods and tools is presented. The main focus is on classifying models. All other issues are in a sense derived from that. The essence of classification is to group in *equivalence classes* according to a number of criteria. A classification may help in choosing the most appropriate formalism when modelling a system. This in turn may help one choose the most appropriate modelling and simulation tool.

One possible classification makes a destinction between *graphical* and *textual* model representations. This is really a tool issue, but the underlying formalism may be more or less amenable to graphical represenation.

A major classification is the distinction between *deterministic* and *stochastic* models. As mentioned before, even stochastic models may be solved by a deterministic simulator.

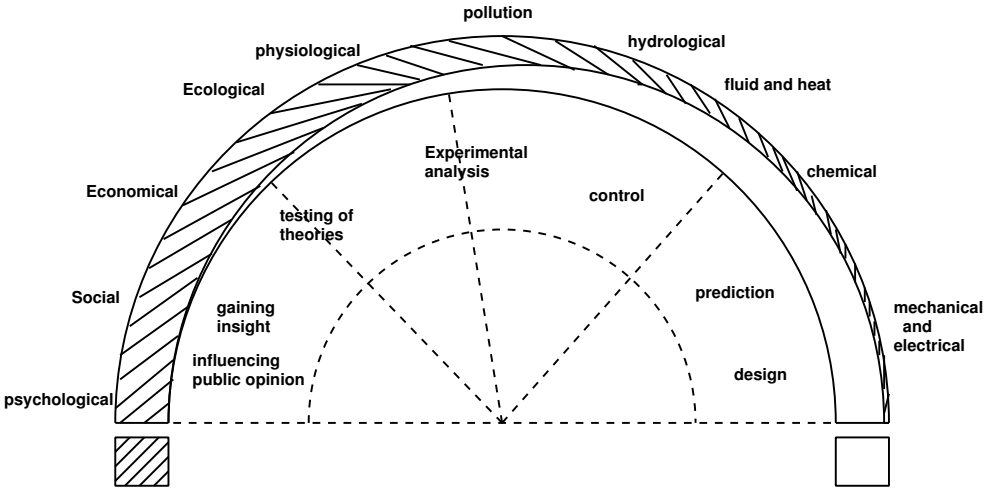

Figure 14: Karplus' classification

7 Application-based classification

The Directory of Simulation Software published yearly by the Society for Computer Simulation is a list of companies and product descriptions. Hundreds are listed, with only a rough, 2-level classification:

- application based: telecom, robotics, resource management, training, process control, power applications, operating systems, networks, manufacturing, industrial engineering, finance, education, chemical, business, biomedical, automotive, batch processes, and aerospace.
- generic: operations research, program generators, graphics, animation, fluid dynamics, discrete event, differential eqation solvers, continuous languages, CAD/CAM, CAE/CASE, and AI/Expert Systems.

This classification does not give much insight in the generic nature of modelling and simulation. However, already, needs and trends can be identified. Most of the software tools are self-contained and closed. To allow exchange and re-use of models as well as inter-operability of simulators, tools must become open and model representation languages standardized.

8 Ill-definedness classification

At a far more abstract level, Karplus' Arch shown in Figure 14 depicts a spectrum of systems ordered according to the amount of *a priori* information we have about them. On the very right, *white box* systems are characterized by complete insight in their working. Based on general governing laws, a model can *deductively* be derived from a priori knowledge. This usually means we build models at the I/O System level. On the very left, *black box* systems are characterized by a complete lack of knowledge about their internal working. Only input-output relationships can be observed. Initially, we can only build models at the I/O Observation Frame level. Using *inductive* techniques, it may be possible to climb up the specification hierarchy, inferring structure from behaviour, to reach higher levels. Often, this is an iterative process whereby a structure is proposed, and after parameter estimation, simulation results are compared to observation data. This comparison (in particular, discrepancies) may yield new insights, which lead to new hypotheses*, etc.* In between is a *grey* area, often called *ill-defined* systems. The study of these systems is most challenging as both observation data and limited a priori knowledge are available. As can be seen in the figure, application areas range from electrical and mechanical systems for white box systems, over biological and physiological models for grey systems, to social and psychological systems for black box systems.

9 System specification classification

In Figure 3, we have shown some common types of segments (trajectories). These may be used to characterize model formalisms.

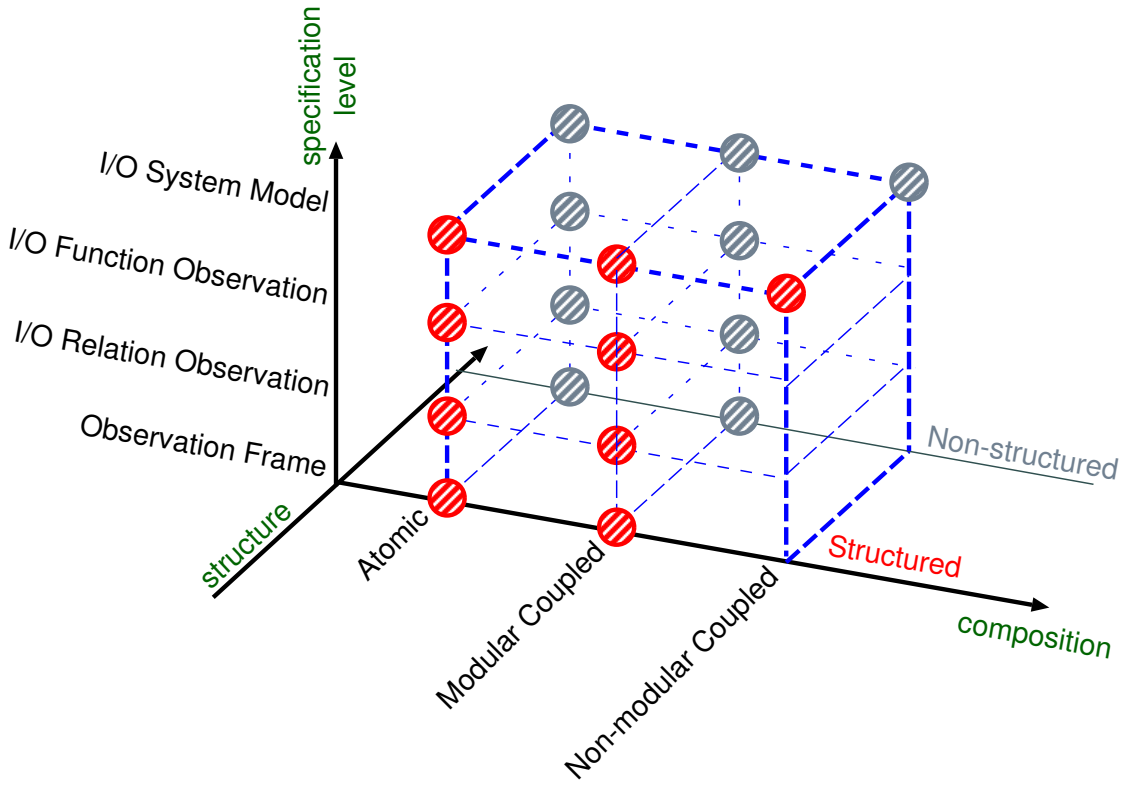

Figure 15: Levels of system specification

	T: **Continuous**	T: **Discrete**	T: {**NOW**}
Q: **Continuous**	DAE	Difference Equations	Algebraic Equations
Q: **Discrete**	Discrete event	Finite State Automata	Integer Equations
	Naive Physics	Petri Nets	

Table 1: I/O system model classification

In Figure 15, we bring together different classifications presented before. The specification level is one dimension of classification. At each level, structured sets may be introduced, to make the specification structured. At each level, and both for structured and for non-structured formalisms, either atomic or coupled models may be constructed. Whereas modular coupled models can be constructed at each level of the specification hierarchy, non-modular coupled models only make sense at the I/O Sytem level.

10 I/O System classification

Many formalisms are situated at the I/O System specification level. For these formalisms, the nature (type) of time base T and state set Q allow for classification. Table 1 shows a few formalisms classified according to the nature of their time base and state set.

11 Discrete event world views

Formalisms belong to the *discrete event* category in case the time base is continuous, but only a finite number of *events* occur in a bounded time-interval, and only at those event times does the discrete state of the system change. Discrete event models are used when

- the behaviour of physical systems is abstracted by means of time-scale or parameter abstraction [MB01]. In many

Figure 16: Car suspension, physical view

cases, this leads to queueing models.
- non-physical systems such as software are studied.

Traditionally, several *world views* have been distinguised:

1. Event Scheduling
2. Activity Scanning
3. Three Phase Approach
4. Process Interaction

12 Tool-oriented classification

In the following, a classification is presented which takes into account the relationship between modelling formalisms and representation languages on the one hand and their *applicability* on the other hand. Here, a modelling and simulation tool user's point of view is taken. This is done by means of a simple example, demonstrating different model representations and their respective merits and drawbacks. In its original form, this is due to Françis Lorenz, during Simulation in Europe discussions [VV96, KVVG94, KVVG95].

The fundamental principle of simulation code generators is to start from a description of a model that is to be simulated and to produce the corresponding simulation program in a –to the user– automatic way. The comparative study of modelling formalisms and corresponding representation languages is therefore of great importance as the language implemented in a particular code generator, with its advantages and flaws, will determine the features of the generator.

The models considered here assume a hypothesis of parameter aggregation (lumped parameter assumption). In mathematical terms, they are represented by differential algebraic equations with a single independent variable, time.

12.1 Physical view

The problem we will discuss is a car suspension (Figure 16), studied here in one dimension. This implies a model of a quarter of the vehicle, because we only study one of the wheels, supporting one quarter of the weight. We assume all elements are linear as it is not possible to represent non-linearities in some of the formalisms we will discuss. Also, we will not represent the lifting of the tire from the road which would introduce a structural discontinuity which could only be represented in a *hybrid* formalism.

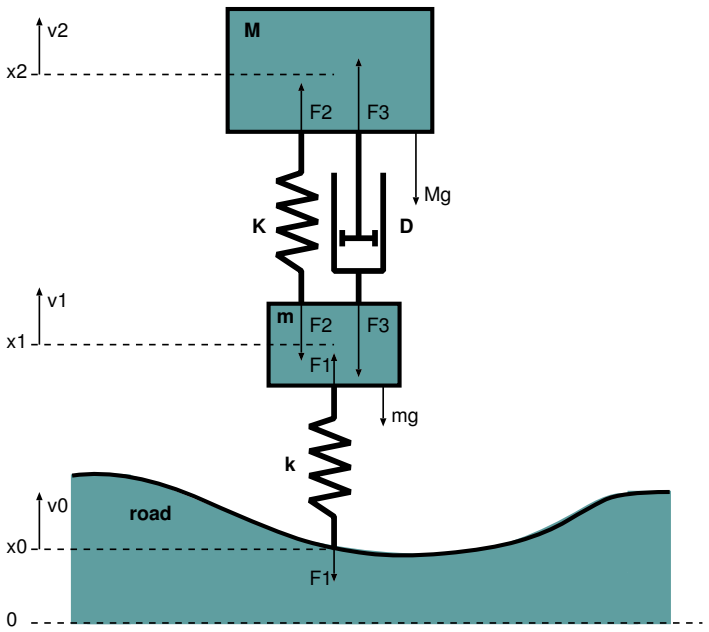

Figure 17: Car suspension, mechanical view

12.2 Physical domain specific views

This example is chosen in the domain of mechanics. A mechanical *concept schema* can be drawn as shown in Figure 17. One could also consider an electrical equivalent of this system as shown in Figure 18.

The analogy chosen is *force* = *current* ("through" variables), *velocity* = *voltage* ("across" variables). In this case, masses become capacitors and springs, self-inductances. The analogy *force* = *voltage*, *velocity* = *current* could also be used (as shown in the following table). Masses are then self-inductances and springs, capacitors; one also has to invert serial and parallel connections. We will come back to this analogy duality.

The analogy between diffent physical domains is summarized in Table 2.

Note that the concept schema is a language for system description. It is not without *ambiguities* (for example, the same zig-zag symbol is used to represent a spring in the context of mechanics and a resistor in the context of electricity) but it is by far the most intuitive. One can express virtually anything in this language, although some uncertainty remains in the exact interpretation of the graph. This language is therefore not totally rigourous and complete. In this respect, where does the schema say that all elements are linear ? The computational causality need not be specified, but there is still much work left to do before simulation code can be generated. In particular, transforming a concept schema into a mathematical formulation is not straightforward.

12.3 Equations

The most "natural" (or at least the one that is designated as such, for simulation purposes) method to present our suspension system is simply by writing down the mathematical equations.

$$\begin{cases} F_1 = k.[I-(x_1-x_0)] \\ F_2 = K.[L-(x_2-x_1)] \\ F_3 = -D.(v_2-v_1) \\ \\ \ddot{x}_1 = (F_1-F_2-F_3)/m \\ \ddot{x}_2 = (F_2+F_3)/M \end{cases}$$

One sees the system is of the fourth order (two second order equations). Note that the position (x_0) is unknown. Also note that it has not been explicitly stated that v_1 and v_2 are the first derivatives of x_1 and x_2. Though this relationship may seem

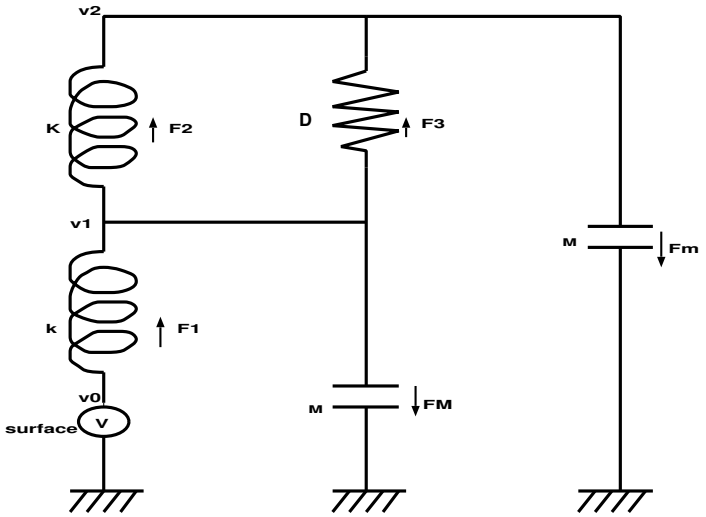

Figure 18: Electrical analogy of car suspension

Physical domain	Effort e	Flow f	Momentum p	Displacement q
Electrical	Voltage u [V]	Current i [A]	Flux Φ [V s]	Charge q [A s]
Translational	Force F [N]	Velocity v [m s^{-1}]	Momentum I [N s]	Displacement x [m]
Rotational	Torque T [N m]	Angular Velocity ω [rad s^{-1}]	Angular Momentum L [N m s]	Angle ϕ [rad]
Hydraulic	Pressure p [N m^{-2}]	Volume Flow q [m^3 s^{-1}]	Pressure Momentum Γ [N m^{-2} s]	Volume V [m^3]
Thermodynamical	Temperature T [K]	Entropy Flow $\frac{dS}{dt}$ [W K^{-1}]	—	Entropy S [J K^{-1}]

Table 2: Physical analogy

obvious to a human, a code generator will only be able to infer this relation with difficulty. To the generator, v_1 and v_2 will be "not computed" (reflected by issuing an error message) or will be considered as unknowns.

Even an example as simple as this can be represented in various ways depending on the analyst and his "style".

Below, another representation of the problem is presented, which uses other variables. Note that now the velocity (v_0) is unknown. The system is of course still of fourth order, but each of the four variables chosen is only integrated once (the accelerations of the two masses and the velocities of compressing the two springs), contrary to the integration of two times two variables (the accelerations of the two masses). This formulation is more directly usable for simulation.

Obviously, this formulation does not improve readability (and hence, re-usability) of large models.

$$\begin{cases} F_1 = k.\Delta I \\ F_2 = K.\Delta L \\ F_3 = D.\Delta v_{12} \\ F_m = F_1 - F_2 - F_3 \\ F_M = F_2 + F_3 \\ \dot{v}_1 = F_m/m \\ \dot{v}_2 = F_M/M \\ \Delta v_{01} = v_0 - v_1 \\ \Delta v_{12} = v_1 - v_2 \\ \Delta \dot{I} = \Delta v_{01} \\ \Delta \dot{L} = \Delta v_{12} \end{cases}$$

Using equations, one can express a plethora of laws of behaviour, but discontinuities are not easily representable in this "language". The language is causally free (non-causal), but the re-use of sub-models requires an almost complete re-examination of each new situation.

At the mathematical level, it is possible to write equations which are not physically meaningful (*e.g.,* do not conserve energy, or lead to negative concentrations).

12.4 State equation

The second set of equations leads us to state equations (state-space form). The four variables to integrate are called state variables and are organised into a vector (in fact, a column matrix). The equations are then written in matrix form.

$$\dot{\underline{X}} = \underline{A}.\underline{X} + \underline{B}.\underline{U}$$

$$\underline{A} = \begin{bmatrix} \frac{-D}{m} & \frac{D}{m} & \frac{k}{m} & \frac{-K}{m} \\ \frac{D}{M} & \frac{-D}{M} & 0 & \frac{K}{M} \\ -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix} \quad \underline{X} = \begin{bmatrix} v_1 \\ v_2 \\ \Delta I \\ \Delta L \end{bmatrix} \quad \underline{B} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \underline{U} = [v_0]$$

This time, the language is causally oriented. We again find the previous characteristics, however amplified by a stricter environment. However, the state equation is relatively easy to solve (by matrix multiplication), except in non-linear cases (the matrices depend on the state vector). Furthermore, it is possible to analyse the eigenvalues of the matrix $\underline{A}$, which yields valuable information about the system's dynamics.

As at the mathematical level, it is possible to write state-space equations which are not physically meaningful.

12.5 Algorithm

We can express the model in the form of a directly usable algorithm. The algorithm below is the transcription in FORTRAN of the above equations.

```
C---+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
C
```

```
      SUBROUTINE SUSPEN ( V0,
     +                    V1, V2, DL01, DL12,
     +                    V1D, V2D, DL01D, DL12D,
     +                    K01, K12, D12, M1, M2 )
C
C INPUT VARIABLE, STATE VARIABLES
      REAL*8 V0,
     +       V1, V2, DL01, DL12
C
C OUTPUT VARIABLES
      REAL*8 V1D, V2D, DL01D, DL12D
C
C PARAMETERS
      REAL*8 K01, K12, D12, M1, M2
C
C INTERNAL VARIABLES
      REAL*8 DV01, DV12, F1, F2, F3
C
C-------------------------------------------------------------------------
C
      DV01 = V0 - V1
      DV12 = V1 - V2
C
C SPRINGS, SHOCK DAMPING
      F1 = K01 * DL01
      F2 = K12 * DL12
      F3 = D12 * DV12
C
C MASSES
      V1D = (F1 - F2 - F3) / M1
      V2D = (F2 + F3)      / M2
C
      DL01D = DV01
      DL12D = DV12
C
      RETURN
      END
C
C-------------------------------------------------------------------------
```

It is clear that one may describe behaviour laws in algorithmic form, but, as with mathematical and state-space models, no fool-proof mechanism is present to guard one from writing physically meaningless models. It is therefore a dangerous expression form. Moreover, it is causally oriented. The current practice of presenting a library of FORTRAN sub-programs as a library of sub-models is therefore to be avoided. Also, this representation does not allow for symbolic analysis of the model.

12.6 Transfer function

Another fashionable way of expression is the transfer function, obtained by the Laplace transform of the equations. They are written here for an unknown velocity; one could have written them for an unknown position as well.

$$\frac{v_1}{v_0} = \frac{k(Ms^2 + Ds + K)}{mMs^4 + D(M+m)s^3 + (mK + MK + Mk)s^2 + kDs + kK}$$

$$\frac{v_2}{v_0} = \frac{k(Ds + K)}{mMs^4 + D(M+m)s^3 + (mK + MK + Mk)s^2 + kDs + kK}$$

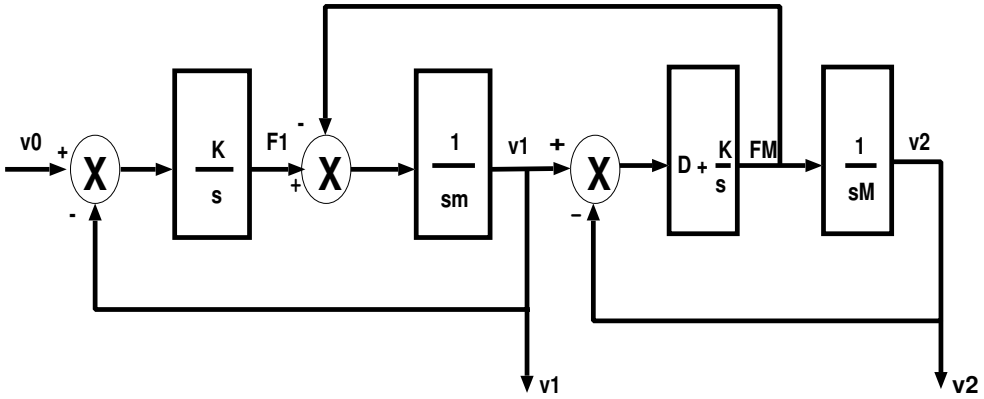

Figure 19: (Transfer Function) block diagram

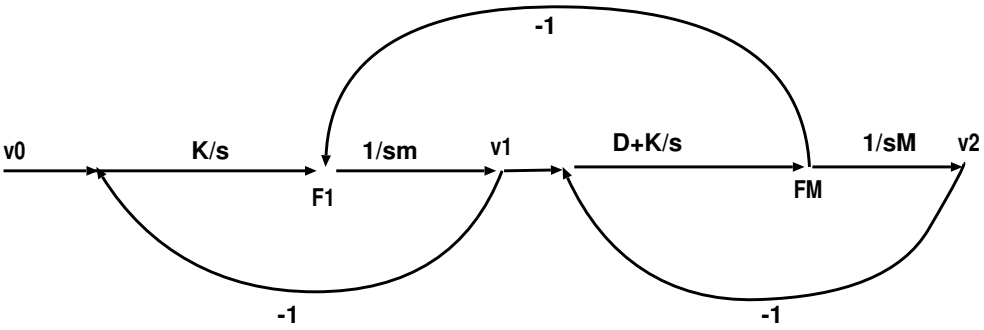

Figure 20: Signal flow graph

This way of representation is only usable in linear cases, which strongly limits its applicability. The readability of the formulas is poor and they are causally oriented (the transfer function pushes the input-output view of the system to its extreme). They are re-usable in a certain range, but most of the time it is easier to re-study a deviating system than it is to modify preceding work. Nevertheless, the utilities developed by control-specialists for this formalism are numerous and extremely useful (*e.g.,* the study of dynamics and stability on Bode, Nyquist or Black graphs, the location of Evans poles*, etc.*), which largely justifies its use in the context of process control problems.

12.7 Block diagram

The *causal block diagrams* were initially nothing but the graphical representation of the transfer functions (Figure 19).

As a graphical representation of the transfer functions, block diagrams strongly improve readability, but leave the other characteristics unchanged. Nevertheless, when used in the *temporal domain*, they become a graphical representation of equations, with all advantages involved. Unfortunately, they also impose a computational causality (as in Matlab/Simulink) on the equations, decreasing their re-usability.

If block diagrams are given a non-causal semantics, this disadvantage vanishes. *Non-causal block diagrams* are called *generalised* block diagrams by Cellier [Cel91].

12.8 Signal flow graph

The signal flow graph is another graphical representation of the transfer functions (Figure 20). It is therefore very similar to the block diagram.

The signal flow graph is only used in the Laplace plane, and is therefore limited to linear systems. Its characteristics are more or less those of transfer functions. It however adds to the arsenal of analysis tools, like the Mason formula, based on a study of the system topology.

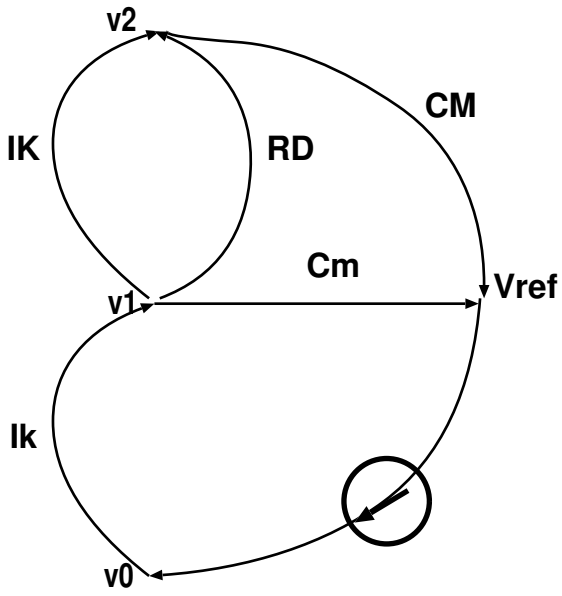

Figure 21: Linear graph

12.9 Linear graph

The linear graph is in a way an equivalent electrical schema of the system, relieved of all electrical connotations. It it is sufficient to compare the linear graph model of our car suspension in Figure 21 to the equivalent electrical concept schema shown above. This method is applicable in linear as well as in non-linear cases. Its principal advantage lies in the fact that it allows for a causally free representation and it also proposes an automatic method for attributing computational causalities called "normal tree search", based on the topological analysis of the model to allow for automatic code generation. In other words, it allows for the creation of real libraries of sub-models and also delivers the necessary tools to use them.

Note that this topological analysis algorithm does not only determine causalities. It also allows the detection of certain modelling errors and certain numerical difficulties (algebraic loops, dependent state variables). These possibilities are present because the language is based on physical concepts (potentials, fluxes, inertial effects, capacitative, dissipative) instead of mathematical concepts (variables, operators). In the model presented here, we have retained the analogy *force* = *flux* (current) and *velocity* = *potential* (voltage). We could just as well have chosen the dual analogy.

12.10 Bond graphs

In the linear graph method, the "parallel" and "serial" links (note that these notions are not as trivial to apply in the mechanical domain) are represented by combining the graph elements serially or in parallel. The bond graph formalism [Cel91, Bro90, Bre84, LW95, SB95] is in principle very similar to the linear graph formalism. In particular, it is based on the same physical concepts, but *all* aspects of the problem are represented here as graph nodes, including its topology: the "0" nodes correspond to "parallel" connections and the "1" nodes correspond to "serial" connections (Figure 22).

This method yields a real formalization of the circuit topology. The structure of the bond graph represents the *physical meta-structure* of the system under study, different by nature from the *technological structure* of the assembly of system constituents. Once the principles of the language have been assimilated, it allows one to understand and master the most intricate physics phenomena (albeit under the lumped parameter assumption).

This time we have abandoned the analogy *force* = *flux* and *velocity* = *potential*, in favour of the analogy *force* = *potential* and *velocity* = *flux*, more common in bond graphs (due to their origin in mechanical engineering). Masses are then inertial elements and springs capacitative elements. This duality of analogy, already mentioned, should not trouble the analyst. In particular, the dualisation of a bond graph is extremely easy: it suffices to replace all elements by their duals ($I \leftrightarrow C, O \leftrightarrow 1, \cdots$) but the meta-structure of the model stays unchanged!

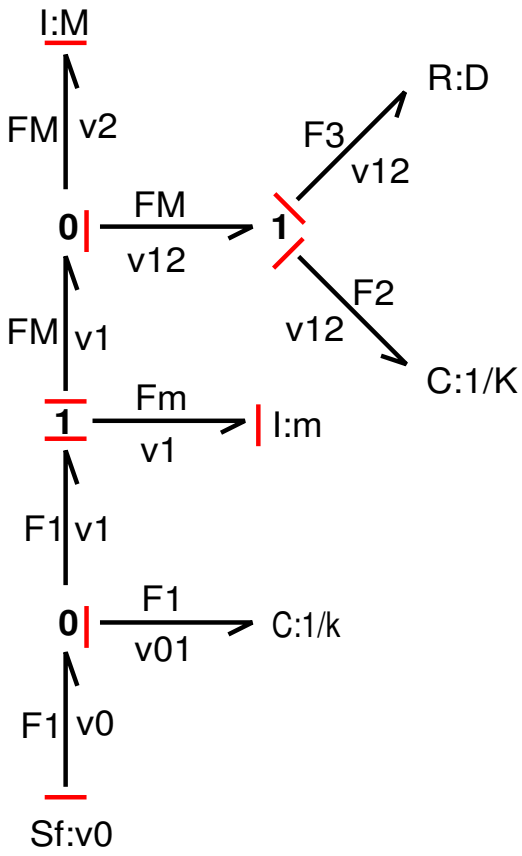

Figure 22: Causal Bond Graph

This method, like the linear method, is applicable in linear as well as in non-linear cases. The representation it allows for is also causally free and it proposes an automatic method of attributing computational causalities (equivalent to the linear method) as well, which also detects the same modelling errors and numerical difficulties.

12.11 Comparison

In Table 3, the formalisms (representation languages) presented are compared in a more structured way, on the basis of the following criteria:

- *Popularity*: general knowledge of the method with engineers and scientific researchers.
- *Readability*: ease with which someone familiar with the method can understand models composed by someone else.
- *Expressiveness*: possibility to describe non-linear systems and to use complex functions (pure delay, tabulated functions, ...) or other special functions (dead zone, hysteresis, ...).
- *Discontinuous models*: possibility to describe functional and structural discontinuities.
- *Structured variables*: possibility to represent model variables (elements accessed by name) and matrices (elements accessed by index) in structures.
- *Modularity*: possibility to hierarchically decompose models into sub-models.
- *Re-usability*: possibility to re-use previously composed models in the context of different problems.
- *Adaptability*: possibility to modify models previously constructed to create a variant of the model (without drastically changing the model).
- *Numerical analysis*: existence of means of analysis linked to the method and yielding information of numerical nature on the model studied.
- *Topological analysis*: existence of means of analysis linked to the method and yielding information of topological nature on the model studied.

12.12 Formalism/language levels

The comparison makes explicit that none of the languages is universally ideal. Each language has its advantages and its flaws. Moreover, for those of the languages for which we have found many flaws and only a few advantages, it may very well be that those advantages are of such importance (large weight factor) that they justify the use of the language in their own right. This is for example the case with transfer functions, which have few advantages according to our criteria, but which are best suited for the study of process control, due to the powerful analysis methods available.

It would thus be useless to search for the "best" language. This does not exist, or rather each language could reveal itself as the best in the context of a specific problem for a specific class of users.

Instead of the segregational view of "better" and "worse" languages, we prefer a more qualified view in which one looks for the optimal use that can be made of these ways of representation. This has led to the classification of the languages using five levels, each level built on top of lower levels, but each level having its use with respect to the problem posed (Table 4). I/O data refers to trajectory data generated from simulations. The algorithmic levels refers to causal models. The mathematical level refers to non-causal models. The physical level refers to non-causal models with physical properties (such as energy conservation) built into the formalism. The technological level refers to representations incorporating technological information which may not reveal any information about the physics of the system.

On the basis of this reflection, one can envision a multi-formalism modelling and simulation environment, in which the user has a choice of formalisms (and corresponding languages) depending on the sub-model he wishes to construct and in which he can freely assemble sub-models. This environment would be able to use one of the "solvers" currently available as a numerical kernel.

The classification of Table 4 is presented in three dimensions in Figure 23. It shows the position of different languages/formalisms with respect to application domain, formalism class (paradigm), and description level.

	Popularity	Readability	Expressiveness	Discontinuous models	Structured variables	Modularity	Re-usability	Adaptability	Numerical analysis	Topological analysis
Concept schema	⊕	⊕	+	+	+	⊕	⊕	+	⊖	⊖
Equations	⊕	-	+	-	+	+	□	□	⊖	⊖
State equations	+	-	□	⊖	+	□	-	-	⊕	⊖
Algorithms	+	-	⊕	□	⊕	□	□	-	⊖	⊖
Transfer functions	⊕	-	-	⊖	⊖	□	□	-	⊕	⊖
Block diagrams	⊕	□	+	□	+	⊕	□	+	+	-
Signal flow graphs	-	□	-	⊖	-	+	□	□	+	+
Linear graphs	+	+	+	-	-	+	+	⊕	□	⊕
Bond graphs	-	+	+	-	+	⊕	⊕	⊕	□	⊕

Legend: ⊖: very bad; −: bad, □: neutral; +: good; ⊕: very good

Table 3: Comparison of representation formalisms/languages

Level	Formalism
Technological	Concept schemas
Physical	Linear graphs Bond graphs
Mathematical	Equations State equations Transfer Functions Non-causal block diagrams Flow graphs
Algorithmic	Causal block diagrams CSSLs/DSblock FORTRAN/C
I/O data	Trajectory data

Table 4: Classification of representation formalisms/languages

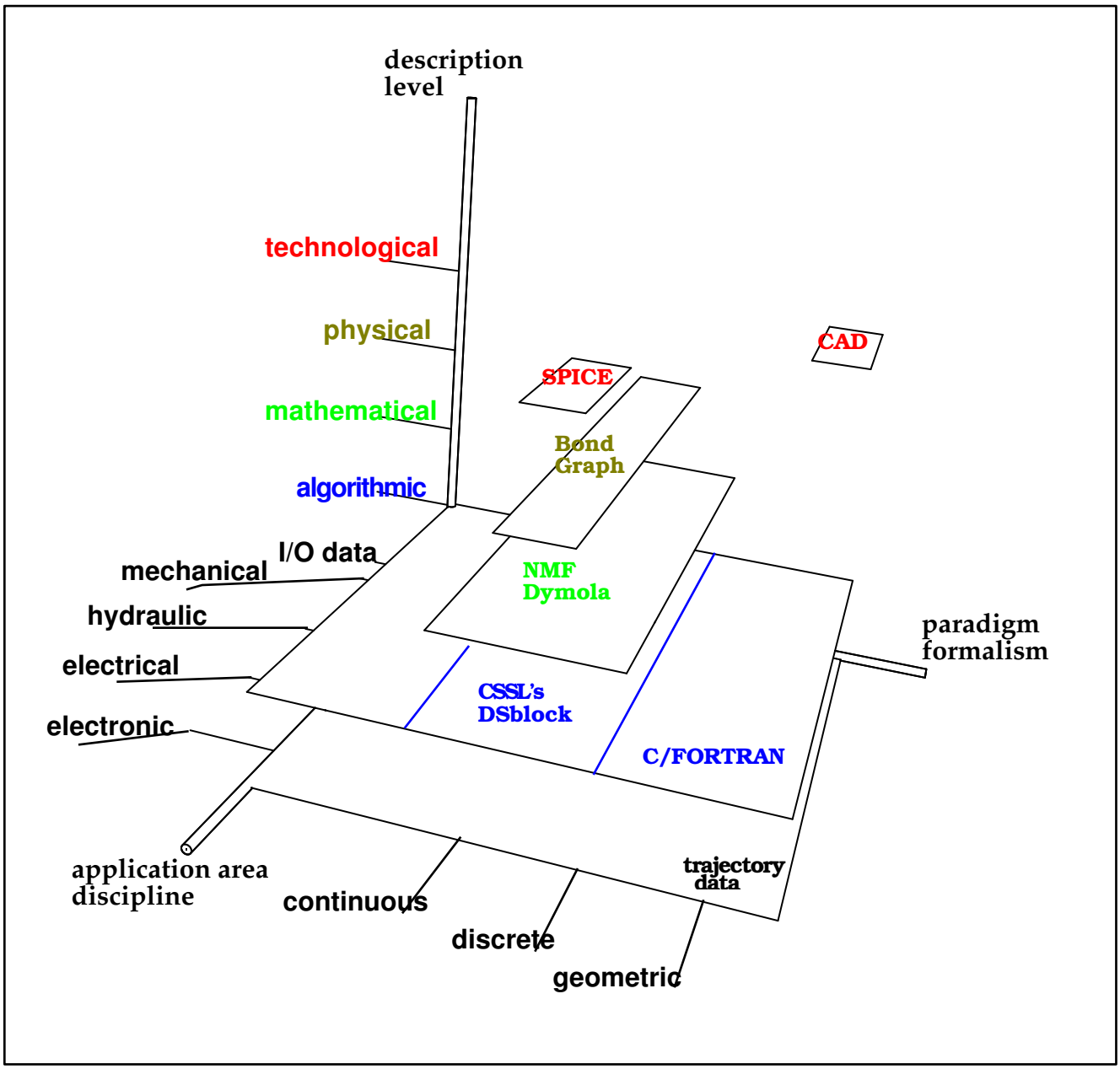

Figure 23: Description level/formalism/application domain classification

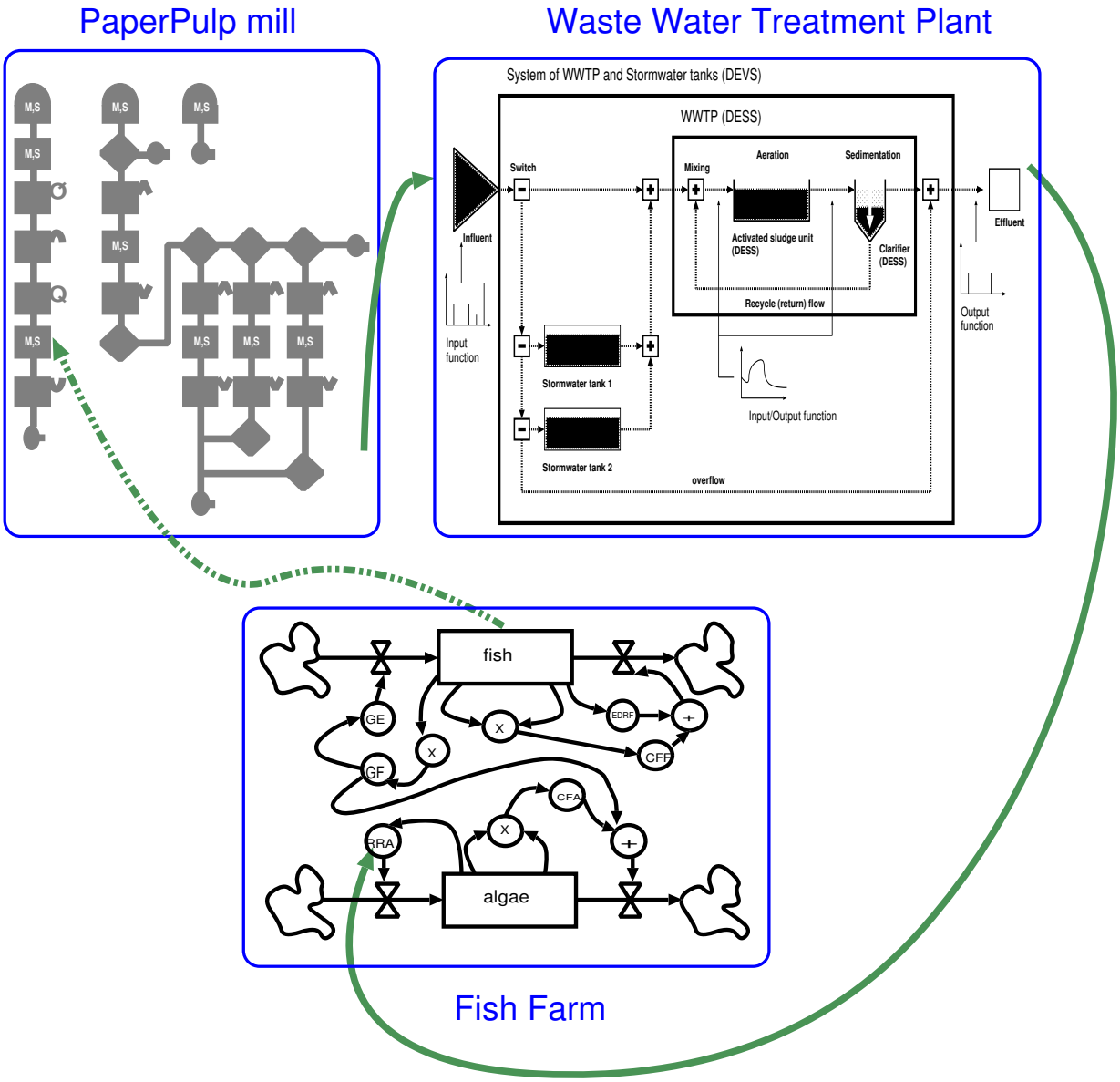

Figure 24: Complex system example

13 Multi-formalism modelling

Though coupled models were introduced earlier, we implicitly assumed that the components were all described in the same formalism. Furthermore, to meaningfully associate behavioural semantics to a coupled model, closure under coupling of the components' formalism was assumed. In *complex* systems, this assumption may no longer be true. These systems are characterized, not only by a large number of components, but also by the diversity of the components. One of the observations of the European Commission's ESPRIT Basic Research Working Group 8467 [VV96] "Simulation for the Future: New Concepts, Tools and Applications" was that for the analysis and design of such complex systems, it is no longer sufficient to study the diverse components separately, using the specific formalisms these components were modelled in. Rather, it is necessary to answer questions about properties (most notably behaviour) of the whole system.

To focus the attention, Figure 24 gives an example of a complex system. The complexity lies in the diversity of the different components, both in abstraction level and in formalism used:

- A paper and pulp mill produces paper from trees with polluted water as a side-effect. This system is modelled as a process interaction discrete-event scheduling system (in particular, in GPSS [Gor96]).

- A Waste Water Treatment Plant (WWTP) takes the polluted effluent from the mill and purifies it. Some solid waste is taken to a landfill whereas the partially purified water flows into a lake. This system is modelled using Differential Algebraic Equations (DAEs) describing the biochemical reactions in the WWTP.
- A Fish Farm grows fish in the lake which feed on algae which are highly sensitive to polluted water. The water is also used for a tree plantation which supplies the paper mill. This system is modelled using the System Dynamics formalism. The dotted feedback arrow from the fish farm to the paper mill indicates the possible disastrous impact of poisoned fish on the productivity of workers in the mill.

It is obvious that decision-making for this system will require understanding of the behaviour of the overall system. Studying the individual components will not suffice. The complexity of this system and its model is due to

- the number of interacting, coupled, concurrent components. Complex behaviour is often a consequence of a large number of feedback loops.
- the variety of components such as software/hardware, continuous/discrete.
- the variety of views, at different levels of abstraction.

A model of a system such as the one described above may be valid (within a particular experimental context) at a certain *level of abstraction*. This level of abstraction, which may be different for each of the components, is determined by the available knowledge, the questions to be answered about the system's behaviour, the required accuracy of answers, *etc.* Orthogonal to the choice of model abstraction level is the selection of a suitable *formalism* in which the model is described. The choice of formalism is related to the abstraction level, the amount of data that can be obtained to calibrate the model, the availability of solvers/simulators for that formalism as well as to the kind of questions which need to be answered. Milner, in his Turing Award lecture [Mil93] rejects the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent systems modelling. Rather, many different levels of explanation, different theories, languages are needed. We believe this view is amplified when arbitrarily complex systems are studied. In our example, different formalisms are obviously used.

As an introduction, to the semantics of multi-formalism models we present the Formalism Transformation Graph (FTG) in Figure 25. The nodes in the graph are formalisms. The horizontal line at the bottom denotes the I/O Observation Relation at which trajectories can be described. The vertical dotted arrows denote the availability of a simulator for a formalism, producing behaviour from a model in that formalism. Other arrows indicate the existence of behaviour-conserving formalism transformations. This will be detailed later. The vertical dashed line demarcates continous model formalisms (on the left) from discrete model formalisms (on the right). The FTG shows a plethora of formalisms, indicating that in general, many classifications are possible. It suffices to annotate the nodes in the FTG with attributes (possibly derived from the formalism structure) and determine equivalence classes based on those attributes.

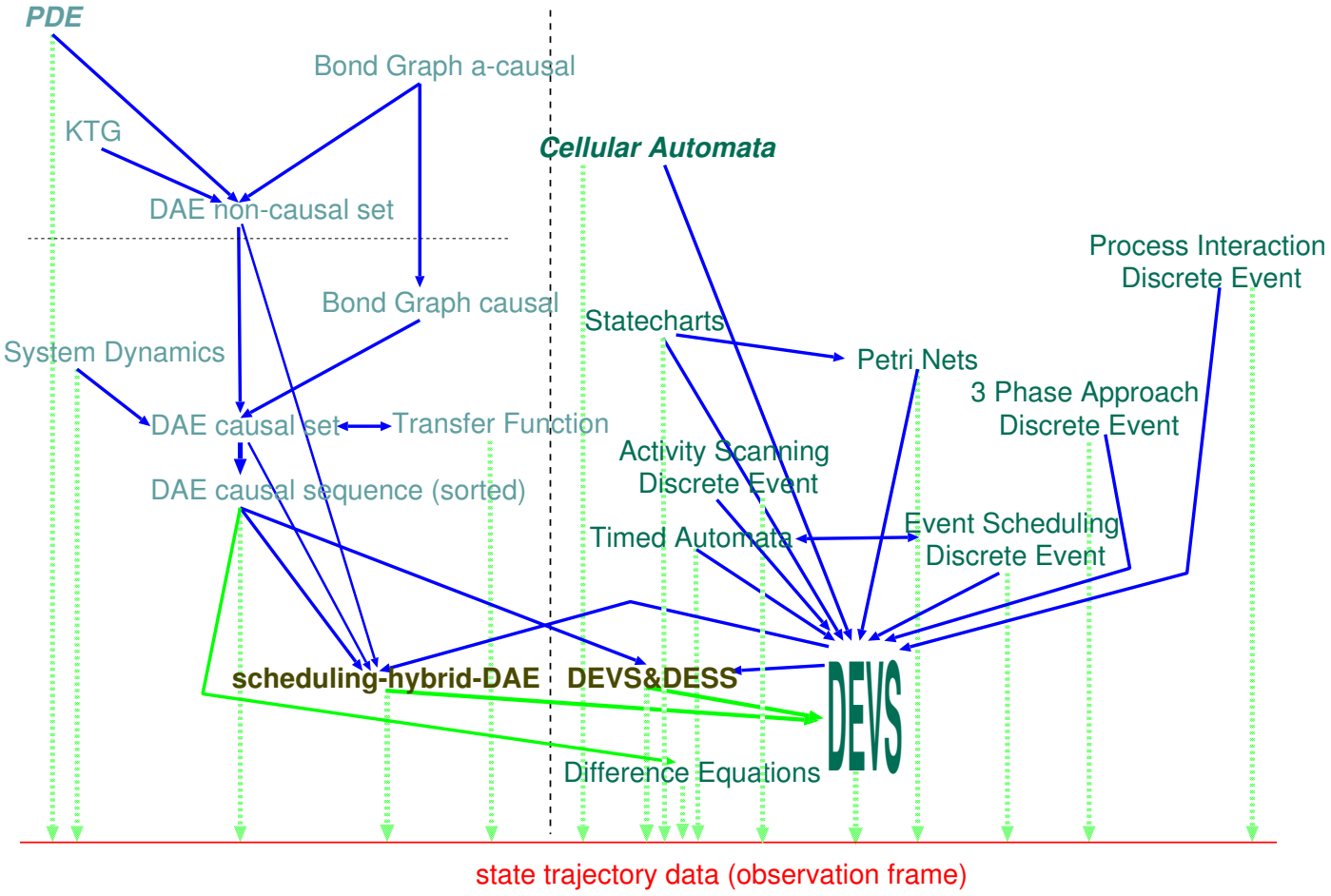

Figure 25: The Formalism Transformation Graph (FTG)

References

[Boo98] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Object Technology Series. Addison-Wesley, second edition, 1998.

[Bre84] Peter C. Breedveld. *Physical Systems Theory in Terms of Bond Graphs*. PhD dissertation, University of Twente, Enschede, Netherlands, 1984.

[Bro90] Jan F. Broenink. *Computer-Aided Physical-Systems Modeling and Simulation: a bond-graph approach*. PhD dissertation, University of Twente, Enschede, The Netherlands, 1990.

[Cas93] Christos G. Cassandras. *Discrete Event Systems*. Irwin, 1993.

[Cel91] François E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, 1991.

[CL89] John Carroll and Darrell Long. *Theory of Finite Automata*. Prentice Hall, 1989.

[Cla92] Filip Claeys. *HGPSS: Object-georiënteerde "process-interaction" Simulatie*. Master's thesis, University of Ghent (RUG), Department of Applied Mathematics, Biometrics and Process Control (BIOMATH), Coupure links 653, B-9000 Gent, Belgium, June 1992.

[CS92] Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, 2(2):109–129, April 1992.

[Gor96] Geoffrey Gordon. *System Simulation*. Prentice Hall of India, second edition, 1996.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Computer*, pages 31–42, 1997.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[Kli85] George J. Klir. *Architecture of Systems Problem Solving*. Plenum Press, 1985.

[KVVG94] Eugène J.H. Kerckhoffs, Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Philippe Geril. Improving the modelling and simulation process. Progress Report 1994:1, ESPRIT Basic Research Working Group 8467, University of Ghent (RUG), Department of Applied Mathematics, Biometrics and Process Control (BIOMATH),Coupure links 653, B-9000 Gent, Belgium, 1994.

[KVVG95] Eugène J.H. Kerckhoffs, Hans L. Vangheluwe, Ghislain C. Vansteenkiste, and Philippe Geril. Improving the modelling and simulation process. Progress Report 1995:1, ESPRIT Basic Research Working Group 8467, University of Ghent (RUG), Department of Applied Mathematics, Biometrics and Process Control (BIOMATH),Coupure links 653, B-9000 Gent, Belgium, 1995.

[LK91] Averill M. Law and David W. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 1991.

[LW95] D.A. Linkens and H. Wang. Qualitative bond graph reasoning in control engineering - part i: Qualitative bond graph modeling and controller design. In François E. Cellier and José J. Granda, editors, *1995 International Conference on Bond Graph Modeling and Simulation (ICBGM '95)*, number 1 in Simulation, pages 183–188, Las Vegas, January 1995. Society for Computer Simulation, Simulation Councils, Inc. Volume 27.

[MB01] Pieter J. Mosterman and Gautam Biswas. A modeling and simulation methodology for hybrid dynamic physical systems. *Transactions of the Society for Computer Simulation International*, 18, 2001. (to appear).

[Mil93] Robin Milner. Elements of interaction. *Communications of the ACM*, 36(1):70–89, January 1993. Turing Award Lecture.

[Nan81] Richard E. Nance. The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179, April 1981.

[SB95] J.E.E. Sharpe and R.H. Bracewell. The use of bond graph reasoning for the design of interdisciplinary schemes. In François E. Cellier and José J. Granda, editors, *1995 International Conference on Bond Graph Modeling and Simulation (ICBGM '95)*, number 1 in Simulation, pages 116–121, Las Vegas, January 1995. Society for Computer Simulation, Simulation Councils, Inc. Volume 27.

[Sch74] Thomas J. Schriber. *Simulation Using GPSS*. Wiley, 1974.

[VV96] Hans L. Vangheluwe and Ghislain C. Vansteenkiste. SiE: Simulation for the Future. *Simulation*, 66(5):331 – 335, May 1996.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[Wym67] A. Wayne Wymore. *A Mathematical Theory of Systems Engineering – the Elements*. Wiley Series on Systems Engineering and Analysis. Wiley, 1967.

[Zei97] Bernard P. Zeigler. *Objects and Systems: Principled Design with Implementations in C++ and Java*. Springer-Verlag, 1997.

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, second edition, 2000.