Hierarchical, Component-Based Modeling Using the Cyber-Physical Modeling Language Modelica

Guido Wolf Reichert¹

¹BSL MANAGEMENT SUPPORT, Kiel, Germany

E-Mail: ¹gwr@bsl-support.de

Abstract
To better absorb structural complexity of real world systems, a hierarchical modeling approach is needed. Using the cyber-physical modeling language Modelica, a recursive system of systems approach to system dynamics modeling can be applied with all the benefits a component-based, object-oriented modeling environment has to offer. In this paper we will take a closer look at the free Business Simulation Library for Modelica, which unlike earlier libraries makes use of acausal, physical connectors. In such an environment, modelers typically do not need to write equations, instead they connect pre-built, reusable components with the additional benefit, that connections observe the fundamental distinction between information and material flows in system dynamics.

Keywords: System dynamics, cyber-physical system, object-oriented modeling, Modelica, Business Simulation Library

Introduction

Subsystem diagrams showing the main subsystems and their interconnections provide a convenient means to represent complex systems at a higher level of abstraction in system dynamics (SD), see Morecroft (1982). Typically, the main subsystems will themselves consist of interconnected systems and ideally a hierarchical representation of a system’s structure (Figure 1) is needed to absorb its structural complexity (Myrtveit, 2000). Stafford Beer’s viable system model provides a good example for this use of recursion in organizational cybernetics, see Schwaninger and Ríos (2008).

In a component-based modeling environment, modelers will typically connect pre-built submodels—usually referred to as “components”—to each other and adjust their parameters,
if necessary. In a hierarchical, object-oriented modeling environment, a complex model will furthermore be built in a nested fashion, exemplified by Figure 1, where the diagram for model M₀ will just show three components (M₁, M₂, M₃).

![Figure 1. Schematic diagram of nested model structure with connectors and connections](image)

Components are connected to each other via interfaces (orange dots), which can be thought of as the modeling equivalent to "connectors" for technical systems, e.g., plugs or sockets. In a hierarchical modeling environment, we will not only need to connect components at the same level, e.g., M₁ and M₂, but also connect outside to inside connectors, e.g., the connector for M₂ is connected to an interface of the interior component M₇. The advantage of clear interfaces will be that we might simply replace one submodel with another one, which has compatible interfaces. The far reaching benefits, e.g., modeling efficiency and component reusability, of an object-oriented extension to system dynamics with a hierarchical, component- and interface-based approach has already been fully recognized and outlined by Myrvold (2000).

Unfortunately, fulfilling the promise of object-orientation with regard to reusability of components including inheritance hierarchy and polymorphism for SD appears to be a difficult endeavor, see Sotaquira and Zabala (2004) for a review of object-oriented approaches in SD including Myrvold’s. Elmasry and Größler (2018, p. 473), for example, still complain: “system dynamics modelling software packages are ‘modularity unfriendly’.” In addition, there also seems to be some kind of divide between modeling and simulation in the natural/physical sciences and the social sciences, e.g., SD does not track energy flows simultaneously with material ones, putting it at a disadvantage with regard to modeling physical systems (Cellier, 2008).

In this paper, we will start with a quick introduction to the cyber-physical modeling language Modelica¹ (Modelica Association, 2021) and clarify misconceptions that are present in our field, which appears to have largely ignored Modelica.² We recommend Zimmer (2016) and ultimately Fritzson (2014) for a more thorough introduction to the language and its applications. We will then examine, how hierarchical, component-based modeling can be utilized for SD by embracing acausal (physical) connectors, a core concept of the language, which has not been done in the earlier System Dynamics library (Cellier, 2008) for Modelica. To better understand, how and why acausal connectors are used in Modelica, we will give a short introduction to object-oriented modeling of cyber-physical systems and relate this to the elementary building blocks of SD simulation models. Finally, we will examine, how this object-oriented, component-based modeling approach deviates from conventional modeling approaches. For this purpose,
we will compare illustrative, conventional SD modeling examples with semantically identical ones built in Modelica. In this paper, we will not look at dynamic behavior, instead our focus will exclusively be on different approaches to represent a system’s *structure*. After all, structure precedes behavior.

**From writing equations to connecting components**

It is useful to illustrate the basic syntax of a programming language by starting with a simple “Hello, World!” program. Applied to the dynamical world of SD, this would arguably amount to implementing a model of exponential growth for a population residing in a single stock. Listing 1 shows a *textual* model of world population growth written in Modelica using readily available data (Central Intelligence Agency, 2022). It appears reasonable to assume that this textual model will be understood by SD practitioners easily, once it has been clarified that `der(population)` is a reference to the stock’s first derivative with regard to time, i.e., its net flow. This immediate familiarity highlights the fact of Modelica indeed being a dedicated *modeling language* for dynamical systems.

**Listing 1. Population model**

```model PopulationModel "Simple model of population dynamics"
  // parameters
  parameter Real initialPopulation(unit = "people") = 7.8e9;
  parameter Real birthRate(unit = "1/yr") = 18.1e-3;
  parameter Real mortalityRate(unit = "1/yr") = 7.7e-3;
  // stocks
  Real population(unit = "people");
  // flows & auxiliaries
  Real reproducing(unit = "people/yr");
  Real dying(unit = "people/yr");
  initial equation
    population = initialPopulation;
  equation
    // flows & auxiliaries
    reproducing = population * birthRate;
    dying = population * mortalityRate;
    // net flow for stocks
    der(population) = reproducing - dying;
end PopulationModel;
```

What may not be so obvious is the use of *equations* (LHS = RHS)\(^3\) instead of *assignment statements* (LHS ::= RHS) in Modelica, which is a primarily equation-based modeling language (Fritzson et al., 2020, p. 19). In fact, the `PopulationModel` could be simulated without any problem, if we were to use `1/population = birthRate/reproducing` and `1/population = mortalityRate/dying` to define the flows in lieu of the equations in Listing 1. We could also have
used \( \text{der}(\text{population}) = 0 \) in the initial equation section to set up the model in equilibrium, which in this case would, of course, only find the trivial solution of \( \text{population} = 0 \) of the initial value problem.

Modelica allows the use of algebraic next to differential equations of motion in models. Accordingly, most Modelica environments employ differential algebraic equation (DAE) solvers like DASSL (Petzold, 1982) by default. In this regard, we need to clarify the statement by Sovilj et al. (2021, p. 227) that “a general-purpose and versatile solver can be slow for simple SD problems.” First, problems in SD do not always remain “simple,” e.g., stiffness may be an issue in a model, models may be hybrid. Secondly, while speed might be an issue, convenience as in setting up models in equilibrium and solving algebraic equations within a model need to be considered as well. Thirdly, Modelica as a modeling language is completely agnostic with regard to the method of solution and most environments offer a choice of solvers, including fast and simple fixed-step Euler. We should also mention that modern solvers like DASSL adapt their integration step size during a simulation run conditioned upon a given error tolerance. A more comprehensive benchmark study for different models and solvers might therefore be a worthwhile endeavor for future research; in such a study numerical errors need to be considered complementing speed as to make a performance comparison truly meaningful.

Since a model and the method used to simulate it, i.e., to solve the system of equations, are strictly separated in Modelica, using the equivalent of \( dt \) or \( \text{TIME \ STEP} \) in equations is illegal. Models are always set up in continuous time, compatible with variable-step solvers. Should we wish to address an instance in time or a sequence of instances in time we will need to make use of events, e.g., when \( \text{time} > 2 \) then <statements> or when \( \text{sample(StartTime,samplingPeriod)} \) then <statements>.

![Figure 2. Component-based, graphical model of population growth](image)

Using pre-built components from the free Business Simulation Library (BSL) (Reichert, 2022) for Modelica, an equivalent graphical model for population growth (Figure 2) can be built by dragging & dropping their representative icons onto a canvas (model diagram). In this case, parameters like the fractional rates or the initial value for the stock are defined locally, i.e., within the components, and can be set using windows and dialog tabs in the modeling environment.

A corresponding textual model is automatically built in the background (Listing 2). For example, a modification equation \((\text{initialValue} = 7.8e9)\) is used to initialize the stock \( \text{population} \), an instantiation of the library’s \text{InformationLevel} class. There are no mathematical equations in the model’s equation section, instead connection equations are established by connecting compo-
nent interfaces in the model diagram: `connect(reproducing.massPort, population.inflow)` and `connect(population.outflow, dying.massPort).

Listing 2. Component-based, textual model of population growth

```model PopulationModel_Components "Simple model of population dynamics"
// classes (components) used in the model
import BusinessSimulation.SourcesOrSinks.ExponentialGrowth;
import BusinessSimulation.SourcesOrSinks.ExponentialDecline;
import BusinessSimulation.Stocks.InformationLevel;
...
// modifications for component instances
InformationLevel population(initialValue = 7.8e9) "World population";
ExponentialGrowth reproducing(hasConstantRate = true, fractionalRate = 18.1e-3);
ExponentialDecline dying(hasConstantRate = true, fractionalRate = 7.7e-3);
equation
  // no mathematical equations needed
  connect(reproducing.massPort, population.inflow);
  connect(population.outflow, dying.massPort);
end PopulationModel_Components;
```

Everything is a system

In explaining the semantics of Modelica and the BSL, we could loosely claim that in our models *everything is a system* and that all systems are connected to each other via clearly specified interfaces, i.e., *connectors*. While we simply used orange dots in Figure 1, the basic stock and flow structure—entities are moved from one stock to another at a specified, constant rate—in Figure 3 has four kinds of connectors, shown as blue triangles and purple squares, which are either filled or empty.

![Figure 3. Basic stock and flow structure](image-url)
From a system of systems perspective a stock may be defined as an elementary system\(^9\) that—in a physical sense—stores matter, energy or information ("mass" or simply "matter"). A stock will have two connectors—shown as filled squares—called inflow and outflow, which can be connected to flows in order to receive or supply matter. To make the passive nature of stocks being states more apparent, stocks are colored in red.

A flow on the other hand is an elementary system that pulls or pushes matter from one stock to another at a specified rate, which may be given either as a constant parameter or via an input connector, i.e., as a continuous-time signal. Note, that material connectors (ports) for stocks and flows differ: Flow ports—shown as empty squares (we may think of the cross-section of a pipe)—can only be connected to stock ports, and vice versa. To visually highlight their activating process nature, flows are colored in green.

Information is exchanged using input (filled triangles) and output (empty triangles) connectors. Stock and flow components will use output connectors to report the amounts contained and the rates of flow, respectively. Since converters are elementary systems that exclusively process information, they will only have input and output connectors. While connectors for information exchange clearly show the direction of flow, i.e., causality, material connectors (ports) do not—they are essentially acausal. Since components are pre-built models, it should immediately make sense that causality is pre-determined by the connector used, as we will know whether information is needed as input or provided as output by a component at the time it is built—irrespective of its actual connections within a model. Hence connections in Modelica models are always undirected edges in graph-theoretical terminology.\(^10\)

Acausality, e.g., connecting two systems without a clear indication of the direction of flow, is unusual in SD, which is rather fundamentally tied to causality. But causality—especially when modeling physical systems—impedes reusability of pre-built components, which is one of the great advantages and aims of object-oriented modeling.\(^11\) Indeed, if we think about the physical analogy of tanks and pumps connected to each other, then a tank component will, of course, have pipe fittings that can be connected to either the suction or the discharge pipe of a pump. The reusability of pipe fittings arises from the fact that they do not "mind" the direction of flow within the pipe they are connected to.

It should be noted that the directional indicators shown within a stock’s icon, i.e., gray double arrowheads next to the connectors, are only suggestive; we may certainly connect a bidirectional flow to any such port. It would spoil reusability, were we to need separate stock components with bidirectional connectors. Whenever we have a clear direction of material flow, it will be indicated visually either by a component’s icon—as is the case for the Transition flow $flowAtoB$ in the diagram—or by using small arrows\(^12\) above connections as in Figure 4.

In Figure 4 the basic stock and flow structure has been turned into a hierarchical model, i.e., it has been split into two subsystems, of which only $systemA$ has a flow port; the outflow from $stockA$ now resides inside that subsystem. Abstract symbols on the subsystems’ icons are used to give visual cues as to what structure is hidden within the components. From looking at the diagram (Figure 4) alone, without an indicator of the direction of flow, we would not be able to tell whether $systemA$ pulls material from a materials inventory as in a pull supply chain or whether $systemA$ delivers material to a materials inventory as in a push supply chain setting. This principal ambiguity exemplifies the reusability—or generality—of a system’s architecture, i.e., the choice of its interfaces.
Inspecting the structure within each subsystem in Figure 5 more closely reveals how a subsystem’s outside connectors are connected to corresponding inside connectors of embedded structure. Connections, i.e., connection equations, between outside and inside connectors effectively are treated as if a connection to the outside connector were directly connected to the linked inside connector. For example, \texttt{connect(systemA.outflow, systemB.inflow)} effectively connects systemA’s outflow (systemA.outflow) to the embedded stock’s inflow (systemB.stock.inflow).

We should mention, that systemA has a vector output (indicated by a dot inside the connector symbol), which will report the amount in the stock as well as the rate of outflow from that stock. Also, the initial values for the stocks have now been made parameters at the subsystem-level. Even though these parameters are given identical names, i.e., initialStock, in both

![Figure 4. Basic stock and flow structure as hierarchical system](image)

![Figure 5. Providing and receiving system](image)
subsystems, their full reference makes systemA.initialStock clearly distinguishable from systemB.initialStock.

The cyber-physical modeling language Modelica concisely describes—and documents—how reusable structures, e.g., modules or molecules of structure, are designed and what interfaces they offer for connections. Clearly, subsystems need to have dedicated "physical" connectors as sending goods or receiving orders are different from sending or receiving instantaneous information signals. Furthermore, acausal connections between stock and flow ports in Modelica guarantee mass balance requirements: While rate information from an information output connector can be connected to multiple input connectors, a flow port cannot be connected to more than one stock port. Multiple flows connected to a single stock port will also automatically be netted, i.e., all rates will be summed into a single net flow. Thus, the clear distinction between material and information flows within SD is concisely carried over to an object-oriented, component based modeling environment.\textsuperscript{14}

\subsection*{Everything is a circuit—component-based, cyber-physical modeling in a nutshell}

So far, we have not explained why we only needed a single connection from reproducing to population in Figure 2—after all, there is a minor feedback from stock to flow in exponential growth, which seems to be missing in the diagram. We will answer this question—and explain acausal connection semantics more generally—by taking a short excursion to cyber-physical modeling. This section leans strongly on the works by Czichos (2019), Isermann (2005), and Fritzson (2014), which are recommended for in-depth reading. Where possible, direct references to specific sources have been added.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6.png}
\caption{Principle structure of cyber-physical systems}
\end{figure}

In this paper, we will not make a distinction between “cyber-physical systems” (CPS) and “mechatronic systems.” Instead we will use these terms synonymously, see Czichos (2019, pp. 461-511) for a more concise definition of a cyber-physical system. CPS are integrated
systems that combine mechanical/coupled systems, electronic systems, and control/information theory. The typical structure of CPS is shown in Figure 6 (Isermann, 2005, p. 4; Czichos, 2019, p. 50).

Broadly speaking, technical systems from different physical domains (e.g., mechanical, electrical, magnetic, hydraulic, thermal, and chemical systems) can be understood by using electrical analogies, so that systems from different domains can be modeled as if they were electrical circuits. In such an analogous treatment, spacial dependencies are usually neglected and instead interacting systems with *lumped* parameters are considered, which can be modeled using algebraic and ordinary differential equations.

### Table 1. Energy carriers with associated potential and flow quantities

<table>
<thead>
<tr>
<th>Domain Type</th>
<th>Potential</th>
<th>Flow</th>
<th>Carrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electrical</td>
<td>Voltage</td>
<td>Electric Current</td>
<td>Charge</td>
</tr>
<tr>
<td>Translational 1D</td>
<td>Position</td>
<td>Force</td>
<td>Linear momentum</td>
</tr>
<tr>
<td>Rotational 1D</td>
<td>Angle</td>
<td>Torque</td>
<td>Angular momentum</td>
</tr>
<tr>
<td>Flow, Hydraulic</td>
<td>Pressure</td>
<td>Volume flow rate</td>
<td>Volume</td>
</tr>
<tr>
<td>Thermal</td>
<td>Temperature</td>
<td>Heat flow rate</td>
<td>Heat</td>
</tr>
<tr>
<td>Chemical</td>
<td>Chemical potential</td>
<td>Particle flow rate</td>
<td>Particles</td>
</tr>
</tbody>
</table>

The modern, object-oriented approach to modeling CPSs can be seen as an extension of Henry M. Paynter’s (1961) bond graphs. In bond graphs, physical components are connected to each other via bonds that represent the instantaneous flow of energy or power. Typical technical components—other than mere sources or sinks—can be depicted as a quadripole (Figure 7), i.e., an electrical circuit with four terminals (Czichos, 2019, p. 31). In Modelica an acausal (physical) connector always consists of a pair of connectors: Each port in Figure 7 has one connector for a *flow* variable (e.g., electrical current) and another connector for the *potential* variable (e.g., voltage being *effort* in the electrical domain). Table 1 shows how potential and flow quantities are usually chosen for different physical domains in Modelica libraries (Fritzson, 2014, pp. 745-746).

Technical systems are then described using the following elementary process elements: *sources, sinks, storages, couplers*, and *converters*. For example, *couplers* are responsible...
for the transport of energy, matter or information in a system, while converters change energy input from one domain into energy output of another. Table 2 lists elementary system examples for different domains (Isermann, 2005, p. 45; Czichos, 2019, p. 38).

Table 2. Exemplary process elements in different energy domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Sources</th>
<th>Storages</th>
<th>Couplers</th>
<th>Converters</th>
<th>Sinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mechanical</td>
<td>water reservoir</td>
<td>spring flywheel</td>
<td>lever joint</td>
<td>piston in cylinder</td>
<td>shock absorber</td>
</tr>
<tr>
<td></td>
<td>wind</td>
<td></td>
<td>transmission</td>
<td>aerofoil</td>
<td>friction</td>
</tr>
<tr>
<td>Thermal</td>
<td>solar radiation</td>
<td>thermal capacity</td>
<td>thermal inductance</td>
<td>Peltier device</td>
<td>cold environment</td>
</tr>
<tr>
<td></td>
<td>heat of earth</td>
<td></td>
<td>thermal radiation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thermo-</td>
<td>combustion</td>
<td>gas volume</td>
<td>vaporization</td>
<td>compression</td>
<td>throttling</td>
</tr>
<tr>
<td>dynamics</td>
<td></td>
<td></td>
<td>condensation</td>
<td>expansion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exothermic</td>
<td></td>
<td>condensation</td>
<td>expansion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>processes</td>
<td></td>
<td>condensation</td>
<td>expansion</td>
<td></td>
</tr>
<tr>
<td>Electrical</td>
<td>accumulator</td>
<td>capacitor</td>
<td>el. conductor</td>
<td>piezoelectric actuator</td>
<td>resistance</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>transformer</td>
<td>electric motor</td>
<td>eddy current</td>
</tr>
<tr>
<td></td>
<td>electrical grid</td>
<td>inductor</td>
<td>transformer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chemical</td>
<td>exothermic reaction</td>
<td>accumulator</td>
<td>matter flow</td>
<td>chemical reaction</td>
<td>endothermic reaction</td>
</tr>
<tr>
<td></td>
<td>reaction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Balance equations for stored masses, energies and impulses, constitutive equations, phenomenological equations or entropy balance equations can be formulated according to physical laws and give rise to algebraic or differential equations in Modelica (Isermann, 2005, pp. 34-37). For example, in the electrical domain where we have the electrical current $i$ as flow and voltage $v$ as the potential, components with an in (positive) and out (negative) port—like resistors or capacitors—will have to meet the following equations:

\[
\begin{align*}
     & C.\text{in}.i + C.\text{out}.i = 0 \\
     & C.v = C.\text{in}.v - C.\text{out}.v \\
     & C.i = C.\text{in}.i \\
\end{align*}
\]

Equation (1) demands that what flows in must flow out, while (3) ties the component’s current to the positive (in) port’s current. Equation (2) defines the component’s potential as the difference of potentials at the ports. A simple, idealized resistor may then be defined using Ohm’s law which adds the following equation to the general port equations (1)-(3), where $R$ denotes the component’s resistance:

\[
\begin{align*}
     & C.v = C.R \times C.i \\
\end{align*}
\]
Next to the before mentioned equations, which can be defined for components irrespective of their place in a system, e.g., within a circuit, connecting acausal connectors of two separate components in Modelica automatically generates connection equations. For acausal connectors, these can be seen as generalizations of Kirchhoff’s laws for electrical circuits to other domains (Fritzson, 2014, p. 39). Connecting the out-ports of components \( C_1 \) and \( C_2 \) with the in-port of component \( C_3 \) in Figure 8 establishes what is called a connection set, where each port has potential \( p \) and flow \( f \).

![Figure 8. Potentials and flows in a connection set](image)

For a connection set two types of equations are automatically created. Equation (5) corresponds to Kirchhoff’s first law and guarantees that whatever flows out of a component, i.e., the flow at a component’s port is negative, must enter at the connected port (flow at that component’s port is positive). In other words, if the outflows \( C_1\text{.out}.f \) and \( C_2\text{.out}.f \) are known (i.e., fixed), we will have made sure that the inflow into \( C_3 \) is the sum of these flows simply by connecting the components.

\[
C_1\text{.out}.f + C_2\text{.out}.f + C_3\text{.in}.f = 0 \tag{5}
\]

\[
C_1\text{.out}.p = C_2\text{.out}.p = C_3\text{.in}.p \tag{6}
\]

Kirchhoff’s second law (6) requires that the potential within a connection set remains equal. If the potential is fixed for any of the ports in the connection set(\( C_1\text{.out}, C_2\text{.out}, C_3\text{.in} \)), then it will be fixed for the remaining ports.

Equations arising from component definitions, e.g., (1)-(4), and from connections, e.g., (5) and (6), suffice to define a system of differential and algebraic equations simply by connecting pre-built components in a diagram. With regard to models of CPS, dynamic behavior rather strictly arises from a system’s structure.

### System dynamics from a cyber-physical perspective

In system dynamics, flows are modeled without taking a corresponding potential into consideration (Cellier, 2008). Nonetheless, Kirchhoff’s laws for physical connections can still be applied and thus acausal connectors are used in the BSL for stock ports (Listing 3) and flow...
ports (Listing 4). The potential variable (*stock*), defined as part of these connectors, is simply used to propagate the amount contained in a stock within a connection set.

Since flow ports must be connected to a single stock port, while stock ports cannot be connected to each other, there can be no ambiguity: In any connection set there will be only one stock port. Stock-related classes will be responsible for integrating given flows connected to their ports as to determine the *stock* variable, while flow-related components will set the rates of flows (*rate*). We can now understand, why only a single connection between *reproducing* and *population* is needed in Figure 2 to model exponential growth: The level information is obtained from the *FlowPort* connector. Note, that this additional flow of information is indicated by blue arrows within the components' icons.

**Listing 3. Stock port connector**

```plaintext
connector StockPort "Connector for stock components"
    import BusinessSimulation.Units.Rate;
    Real stock "Current amount of 'mass' in the stock"
    flow Rate rate "Flow that affects the stock"
    output Boolean stopInflow "= true, if nothing can flow into the stock"
    output Boolean stopOutflow "= true, if nothing can flow out of the stock"
end StockPort;
```

Stock and flow port connectors in the BSL are, in fact, *composite connectors* as they have causal connectors as well. Using Boolean flags (*stopInflow*, *stopOutflow*) a stock may signal to any flow, that is connected to one of its ports, whether it can be filled or drained.

**Listing 4. Flow port connector**

```plaintext
connector FlowPort "Used to represent stock and flow connections"
    import BusinessSimulation.Units.Rate;
    Real stock "The current amount of 'mass' in a connected stock"
    flow Rate rate "Flow that affects the stock"
    input Boolean stopInflow "= true, if nothing can flow into a connected stock"
    input Boolean stopOutflow "= true, if nothing can flow out of a connected stock"
end FlowPort;
```

We can categorize the basic building blocks of system dynamics according to the same categories used for technical systems (Table 3). While this, of course, is not a valid classification in a strict physical sense, the analogies may be close enough for descriptive purposes. In most cases, it makes sense to consider the combination of a *cloud*—in the BSL concisely shown as a stock with infinite capacity—and a flow as *physical sources or sinks* next to mere clouds.
Accordingly, processes of growth or decline at a system's boundary, e.g., ExponentialGrowth and ExponentialDecay, are found in the package SourcesOrSinks as well.

<table>
<thead>
<tr>
<th>Table 3. Elementary classes of the Business Simulation Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
</tr>
<tr>
<td>PHYSICAL</td>
</tr>
<tr>
<td>• Matter</td>
</tr>
<tr>
<td>• Energy</td>
</tr>
<tr>
<td>• Information</td>
</tr>
<tr>
<td>CYBERNETIC</td>
</tr>
<tr>
<td>• Information</td>
</tr>
</tbody>
</table>

There are three types of stocks in the BSL. The unrestricted InformationLevel can be drained below zero and filled up to infinity. The MaterialStock carries a plus sign in its icon to indicate that there cannot be negative amounts in the absence of antimatter, while the CapacityRestrictedStock is the only stock that really meets the physical analogy of a bathtub, which cannot be drained below zero, i.e., its minimum capacity, nor be filled above its maximum capacity.

Unidirectional and bidirectional flows (Flows.Unidirectional, Flows.Bidirectional) are listed as couplers. This is debatable, because they are active process elements, i.e., flows can be used as actuators moving matter, energy or information according to rates determined in the cybernetic sections of a model. Interactions (Flows.Interactive) are categorized as physical converters, because outflow and inflow are not directly coupled and there often is a change in units, e.g., broken flows.

The two sensors (FlowPortSensor, StockPortSensor) shown as physical converters in Table 3 below interactions need some explanation. In non-hierarchical system dynamics environments, connecting two or more flows to a stock will introduce the rate variable to the integral equation describing the stock. A concept like measuring a flow (or a stock) is unusual in system dynamics.

When we are designing pre-built components, we will not know the name or number of components that will be connected to a stock or flow port in advance. While we may simply connect an outside stock port to an inside stock port as in Figure 5b to accumulate arbitrarily many flows that are connected to the outside port, how can we obtain the rate information for the incoming flow to process it—likely after smoothing—within an information converter? Sensor components therefore are a rather inevitable addition to system dynamics in a hierarchical, component-based modeling environment.
With regard to information processing in the cybernetic sphere Converters need no further introduction. The policy component symbol \( \text{MoleculesOfStructure.Policy} \) below a converter takes up the suggestion given by Morecroft (1982) to have a converter-like component to clearly indicate decision making processes in diagrams.

The notion of InformationSources fortunately is as plausible as it is technical. ExogenousData as well as typical test signals, e.g., a RampInput, can be found among these classes. The ConstantConverter, listed here below the information sources, has been introduced by Forrester (1961) and for this historical reason is still listed as a converter in the BSL; there is an equivalent component called ConstantInput that fits into the new categorization.

The icon shown in the storage column in the cybernetic sphere corresponds to the one typically used for information delays (e.g., Smooth, DelayFixed). In principle an InformationLevel may certainly have been listed here as well, but the notion of storage in information processing and computation seems different and thus stocks will typically be nested within policy components or information delays—not warranting a listing of their own.

Structure explains behavior—if you can see it

It is a central tenet of system dynamics that a system’s structure causes its behavior in time. Let’s start looking at deviations from conventional SD modeling by putting this tenet to a simple test: Which of the two stock and flow diagrams (Figure 9) is a model of basic first-order stock adjustment? Which of the constants \( C_1 \) or \( C_2 \) can be interpreted as setpoint of the balancing feedback loop (B)?

![Figure 9. Two models that really differ (conventional SD notation)](image)

It becomes immediately apparent that the graphical elements used in stock and flow diagrams are not sufficient to unambiguously explain how rates of flow are determined. Compare the diagrams in Figure 9 with those in Figure 10, which were produced using the BSL and which are semantically identical to the ones shown in Figure 9. Modelica allows each class, e.g., each component in a model, to have an icon of its own. This is most noticeable with regard to converters. We immediately recognize that \( A_1 \) calculates a gap between the setpoint \( C_1 \) and the amount contained in stock \( S \) in Model A. We also see that the output from
$A_2$ is added to the current rate of outflow $\text{OUT}$. In other words, we can verify that Model A is a model of first-order stock adjustment without having to look at equations.

![Diagram](image)

**Figure 10.** Two models that really differ (BSL)

Icons can also reference parameter values: $s$ exhibits the stock's initial value (i.e., the parameter $\text{initS}$), while the constant converters $C_1$ and $C_2$ directly show their numerical values. Icons can further be used to reveal finer details, e.g., the division in Model B ($A_1$) is guarded against division by zero—the one in Model A ($A_2$) is not.

**Dynamic stocks versus flows with hidden stocks**

Modeling delays is of central importance in SD and Forrester (1961) as well as Sterman (2000) dedicate a whole chapter to this important topic in their seminal introductory works. It also marks a central point of deviation for the BSL in comparison with conventional SD. This becomes evident upon scrutinizing Figure 11, which presents two semantically identical models of a basic third-order delay structure with two inflows. We may again point out that the delay is rather hard to spot in Figure 11a and that its specific nature remains opaque, e.g., the type of delay, the average delay time, and (if applicable) its order are invisible in the diagram. More fundamentally, the convoluted causal links in Figure 11a suggest that the structure shown in Figure 11b is missing out in this regard. But is it? Are there good reasons for many SD tools having parted from the basic diagramming suggestion for higher-order delays given by Forrester (1961, p. 88), which is rather similar to Figure 11b?

As we have used exponential delays of third-order in both cases, we are undoubtedly aware of the fact that three stocks have been “hidden” somewhere. In Figure 11a they are hidden within the equation for outflow, i.e., Vensim’s Delay $N$ function, and in Figure 11b they are embedded within the stock, i.e., the BSL's Delay$N$ component.
To clarify how a function like Delay N works in the case of a third-order delay, we can image it as a kind of converter, which takes in information and computes the rate of outflow as its output. In Figure 12 the hidden structure within the delay function is made explicit inside a green box—reminding us of the fact that we are modeling a delayed flow. We have to pass the sum of inflow1 and inflow2 to the function in order to arrive at the delay's total inflow. We also have to pass the initial stock value and the delay time.

It becomes immediately clear that this kind of “double-entry accounting” is a bit tedious and error prone, e.g., it would be nice if inflows were added automatically. If we fill the stock inDelay without passing the rate information on to the Delay N function, any amount entering the stock via such an additional flow will never leave the stock. Draining the stock looks temptingly easy as well, but is actually a bit involved; see Fang and Dangerfield (2004) for how to do this consistently.
Hiding (or nesting) the stocks within an aggregate stock component (Figure 13) appears to be a more natural modeling solution as it simply “encapsulates” an explicit model of a material third-order delay relieving modelers from having to manually add and link flow rates as they will automatically be added according to (5). Another advantage of having explicit interfaces for stock and flow connections is that simple modeling errors can be prevented: Using the Boolean flag of the delay’s inflow connector (stopOutflow = true), the component does not allow the stock to be drained at its designated inflow side.

Figure 13. Explicit structure for third-order delay comparable to BSL's DelayN component

A closer look at the outflow of the delay structure in Figure 13 reveals a different kind of causality with regard to the aggregated stock (Junglas, 2016). If we consider the structure within the red DelayN box as a single stock, then it will be drained by a flow, whose rate is set by the stock itself. Thus the normal “division of labor” for stocks and flows is changed as in this case the stock will cover both tasks: setting the rate and integrating the connected outflow.

This structure can be generalized to the notion of a dynamic stock with a dedicated outflow port (StockPort_Special) catching the eye with its red coloring (see the outflow port of inDelay in Figure 11b). Dynamic stocks contain internal processes that are not made explicit in the model, i.e., they are at a lower level than a subsystem with a stock and a flow port like systemA in Figure 4. Dynamic stocks like DelayN, SimpleConveyor, Conveyor, PureDelay, and Oven can only connect to flows with a matching port (FlowPort_Special); in the library, these currently are the OutflowDynamicStock and SplitOutflowDynamicStock components. Special stock and flow ports use the potential variable within the acausal connector to transmit the rate information from the stock to its outflow. Since we cannot connect other flows than the before mentioned unidirectional outflows, modelers cannot accidentally fill or drain dynamic stocks via the outflow port.

Reaching higher levels of abstraction

Up to this point, we have addressed elements that—once physical connectors are understood—have an immediate counterpart in SD and dedicated SD tools, e.g., stock and flow ports are...
simply interfaces for connecting these elementary building blocks. Using Modelica’s flexibility to define *composite connectors* the BSL provides causal connector classes (StockInfoOutput, StockInfoInput), that will report a *record* of information specific to stocks (Listing 5).

**Listing 5.** Record of stock information

```plaintext
record StockInformation "Record of information that can be collected from a stock"
    import BusinessSimulation.Units.{Time,Rate};
    Real infoLevel "Current Level of stock";
    Rate infoInflow(min = 0) "Current rate of inflow";
    Rate infoOutflow(min = 0) "Current rate of outflow";
    Rate infoNetFlow(min = 0) "Net-rate of flow";
    Time infoMeanResidenceTime "Mean time of residence for elements in the stock (-1 indicates infinite time for zero outflow)";
end StockInformation;
```

Each stock in the BSL has a structural parameter (hasStockInfoOutput) to activate a stock information output connector, i.e., StockInfoOutput is a conditional sub-component of any stock component. In Figure 14, which is an excerpt of the ManagementEmployment example that ships with the library, the AbsoluteSensor component is used to read out the rate of inflow (IN), the amount in the stock (S), the mean residence time (MRT), the net rate of flow (F), and the rate of outflow (OUT). This information is then provided via separate information output connectors.

![Figure 14. Using the AbsoluteSensor component to read out stock information](image)

Next to reading out a whole set of data related to a stock at once, we may also directly connect stock information output and input connectors. In the full ManagingEmployment example (Figure 15) the stock workforce, a DelayN, is directly connected to expertise, which is a HinesCoflow, a variation of the typical coflow structure used in SD (Hines, 2015, pp. 50-53). To establish a coflow we only need to make two connections. Further more, a coflow has the
structural design and appearance of a stock. The linear growth process gainingExperience, which has a constant rate of 1 yr per year, can be directly connected to the inflow port of expertise to model the workforce becoming more capable.

Figure 15. HinesCoflow using stock information connectors in the ManagingEmployment example

The diagram in Figure 15 also demonstrates how the diagramming suggestions by Morecroft (1982) can be taken up using Modelica. The well known first-order stock adjustment structure (Hines, 2015, pp. 38-39) is compactly used within the recruitingPolicy component. As material processes and reservoirs are clearly separated from the cybernetic structure of control—and possibly unintended influences—the basic structure of cyber-physical systems (Figure 6) becomes evident in the diagram.

At an even higher level of abstraction, stock info connectors can be employed to implement quantitative causal loop diagrams (CLD) in Modelica as suggested by van Zijdeveld (2007). Figure 16 shows the HealTheWorld example that ships with the library (Bossel, 2004). The variables (societalAction, environmentalLoad, consumption, and population) are recognizable as simplified stocks (e.g., red boxes) with a single stock port and a stock information output connector. Such high level abstraction indeed allows to model CLD similar to an electrical circuit. In this example, the components in between stocks are elasticities (Elasticities). Accordingly, the balancing loop B1 expresses that any percentage change in population will result in an equally valued percentage change in environmentalLoad, i.e., having the same sign. Conversely, any percentage change in environmentalLoad would then cause a percentage change in population with opposite sign, whose absolute value is just a tenth of the original one.

Figure 17 exemplifies that such a loop introduces simultaneous equations using a similar feedback loop between two stocks A and B, where \( \varepsilon_{B,A} \) denotes the A-elasticity of B and \( \varepsilon_{A,B} \) denotes the B-elasticity of A. In the left hand diagram in Figure 17 we see that, to obtain the
fractional rate of growth for $B$, we have to multiply the $A$-elasticity of $B$ (the elasticity for the arrow from $A$ to $B$) with the percentage change for $A$ ($\frac{\dot{A}}{A}$). The diagram on the right hand side then introduces the reciprocal structure for applying the $B$-elasticity to obtain the rate of growth for $A$. The cycle colored in red is a feedback loop without a stock, i.e., a simultaneous equation.

While it is not impossible to solve simultaneous equations in SD software, it is not easy to do and may sometimes even be impossible. In Modelica environments, the standard DAE solver will solve both algebraic and differential equations in the model—no additional measures have to be taken by the modeler.

In such a high level dynamical model, we can add a delay in the same way a resistor is added in an electrical circuit. Nonlinearity may be introduced by using high level lookup components. Such level of abstraction also appears amenable to metaprogramming, i.e., we may write more or less intelligent programs that will construct dynamic models. It is, for example, straightforward to parse a verbal model consisting of statements like “if $A$ grows by $a\%$, then $B$ will shrink by $b\%$” in order to come up with a quantitative CLD diagram. Since any component in such a simple, preliminary model can at once be replaced by a subsystem with more elaborate structure, the object-oriented approach to SD supports fast and evolutionary, i.e., agile, model development.
Conclusion

Looking back upon the complaints and desiderata expressed by Elmasry and Größler (2018), we can now state—with some conviction—that all of this is precisely addressed by using Modelica for SD modeling as presented here. The wish list proposed in their paper is almost exactly matched by Modelica’s semantics.

Furthermore, we are gaining the full benefits of object-oriented development, including the kind of model reusability that Sotaquira and Zabala (2004) missed in existing object-oriented approaches. Employing Modelica we can: (a) inherit and modify structure to build new components or to store scenarios; (b) easily and thoroughly test partial models; (c) re-use pre-built components to more efficiently—and more reliably—build large-scale models; (d) use structural parameters to build more general components, e.g., switch between using a constant or a continuous-time input; (e) immediately replace model components or subsystems with another variant, since we know that its interfaces match up, e.g., switch between different policies in a model; (f) better document and present our models, as each class is documented separately and links to the documentation of its components—the linked documentation in this paper is a case in point; (g) mix discrete-event and continuous time simulation in our models; (h) have better support for collaborative modeling; (i) use the standardized and industry-level functional mockup interface (FMI) for model exchange and co-simulation.

As we have seen, it is important to have the clear separation between information and material flows carry over to an object-oriented modeling approach. As a cyber-physical modeling language Modelica is uniquely placed to enable this. Regarding speed and versatility, we should mention that Modelica is a clearly specified, open modeling language. There are proprietary as well as open source environments available to build Modelica models and it is presently being implemented using the highly performant Julia programming language (Elmqvist et al., 2021).

From a more idealistic perspective, we feel that Modelica fulfills the promise of systems science in that we may mix different technical domains as well as system dynamics on the
same modeling canvas. It stands to reason that it is easier for a versatile, cyber-physical modeling language to accommodate SD than for dedicated SD environments to accommodate cyber-physical modeling in comparable efficiency and elegance.

Notes

1. Modelica is a registered trademark of the Modelica Association.

2. Robert Powers (2011), to give a striking example, not once mentions Modelica or its free System Dynamics library (published in 2002) in his thesis on object-oriented approaches to managing complexity, which has a whole section on “previous approaches in SD.”

3. Left-hand side (LHS) and right-hand side (RHS).

4. To find a nontrivial solution in more involved models, we would need to "prime" the solver with a nontrivial start value, for example, Real population(start = 10).

5. Algebraic equations in SD are often referred to as simultaneous.

6. Events in Modelica can be time or state events, i.e., an event may be triggered by the amount in a stock crossing some threshold.

7. All Modelica models and diagrams in this document were produced using Wolfram System Modeler Version 13.

8. For clarity, all the components used within the model have been imported from the library first; it is not necessary usually.

9. An elementary system is atomic in the sense that we typically will not be inclined to visually inspect its inner structure, which often consists of explicit equations—not connected subcomponents.

10. Accordingly, order does not matter in a connection equation and connect(a,b) is strictly equivalent to connect(b, a).

11. In cyber-physical models, acausality is more generally tied to using algebraic equations, where it will not necessarily be clear in advance, which variables will be givens and which will be unknowns in need of being solved for.

12. Such visual flow indicators are mere graphical annotations in diagrams.


14. While some SD tools allow modularity and distinguish mass flow connections from information flow connections to the author’s knowledge in all cases Figure 5a would be showing a flow from the stock to a cloud, which is clearly misleading and ultimately
wrong. At the local level of the subsystem there is no stock with infinite capacity. The use of flow ports thus is a more concise way to model material connections between subsystems.

15. The flow is also referred to as the *through variable*.

16. The potential or effort is sometimes referred to as the *across variable*.

17. Couplers are also called *transformers*.

18. The convention here is that the *positive* side is considered to be the *in* port.

19. Composite connectors are aggregations of connectors (Fritzson, 2014, p. 207).

20. Documentation for the package `SourcesOrSinks` and its classes to be found at [https://wolfr.am/131UH4LNB](https://wolfr.am/131UH4LNB)

21. Documentation for `InformationLevel` to be found at [https://wolfr.am/131UUGhnv](https://wolfr.am/131UUGhnv).

22. Documentation for `MaterialStock` to be found at [https://wolfr.am/131UZDtSx](https://wolfr.am/131UZDtSx).

23. Documentation for `CapacityRestrictedStock` to be found at [https://wolfr.am/1325eXQiI](https://wolfr.am/1325eXQiI).

24. Unfortunately, system dynamics as a discipline, while being critical with regard to false mental models, quite often propagates misleading physical analogies: An unrestricted stock in system dynamics is *not* a bathtub, whose capacity is restricted and whose outflow is proportional to its content (Torricelli’s law). A flow is arguably better represented by a *pump*, e.g., a paddle wheel pump, than by a mere valve.

25. Documentation for the package `Flows.Unidirectional` and its classes to be found at [https://wolfr.am/1327wu1NO](https://wolfr.am/1327wu1NO)

26. Documentation for the package `Flows.Bidirectional` and its classes to be found at [https://wolfr.am/1327zvn6d](https://wolfr.am/1327zvn6d)

27. Documentation for the package `Flows.Interaction` and its classes to be found at [https://wolfr.am/13282mnIW](https://wolfr.am/13282mnIW)


29. Documentation for `StockPortSensor` to be found at [https://wolfr.am/1329LMc7Q](https://wolfr.am/1329LMc7Q).

30. Documentation for the package `Converters` and its classes to be found at [https://wolfr.am/132h4UYZW](https://wolfr.am/132h4UYZW)

31. Documentation for the package `MoleculesOfStructure.Policy` and its classes to be found at [https://wolfr.am/132hSdX2c](https://wolfr.am/132hSdX2c)

32. Documentation for the package `InformationSources` and its classes to be found at [https://wolfr.am/132ibgeAL](https://wolfr.am/132ibgeAL)
33. All models in this paper showing conventional SD notation were produced using Vensim DSS 6.3. *Vensim* is a registered trademark of Ventana Systems, Inc.

34. Forrester suggested to include the delayed outflow within the aggregate component representing a delay structure. While this could be easily done in Modelica, it is a deliberate design choice to have the outflow be more visible using elementary components in the BSL.

35. Documentation for DelayN to be found at [https://wolfr.am/130nco99Z](https://wolfr.am/130nco99Z).

36. In Vensim we actually need to pass the initial outflow, which is typically the initial stock value divided by the delay time.

37. In the BSL subsystems with both stock and flow ports are categorized as *transceivers*.

38. Documentation for SimpleConveyor to be found at [https://wolfr.am/130p637Qk](https://wolfr.am/130p637Qk).

39. Documentation for Conveyor to be found at [https://wolfr.am/130p8Do42](https://wolfr.am/130p8Do42).

40. Documentation for PureDelay to be found at [https://wolfr.am/130paNXwe](https://wolfr.am/130paNXwe).

41. Documentation for Oven to be found at [https://wolfr.am/130pcLgjT](https://wolfr.am/130pcLgjT).

42. Documentation for StockInfoOutput to be found at [https://wolfr.am/130xlwkbS](https://wolfr.am/130xlwkbS).

43. Documentation for StockInfoOutput to be found at [https://wolfr.am/130xsE4CK](https://wolfr.am/130xsE4CK).

44. A specialized class of Modelica (Modelica Association, 2021, p. 45).

45. A structural parameter cannot be changed between simulation runs and is typically used to determine the order of a delay or the turn conditional components and their connections on or off—effectively making structural changes to a model before it is compiled for simulation.

46. Documentation for the ManagingEmployment example to be found at [https://wolfr.am/130zZ2vcZ](https://wolfr.am/130zZ2vcZ).

47. Documentation for AbsoluteSensor to be found at [https://wolfr.am/130AvMDnv](https://wolfr.am/130AvMDnv).

48. Documentation for HinesCoflow to be found at [https://wolfr.am/130BUNyUn](https://wolfr.am/130BUNyUn).

49. Documentation for FirstOrderStockAdjustment to be found at [https://wolfr.am/130DVCX0w](https://wolfr.am/130DVCX0w).

50. The model used by van Zijderveld to demonstrate MARVEL is implemented in the library as an example called *SoftwareReleaseProject*.

51. Documentation for the HealTheWorld example to be found at [https://wolfr.am/131usp0V0](https://wolfr.am/131usp0V0).

52. Documentation for Elasticity to be found at [https://wolfr.am/131vDs5Ca](https://wolfr.am/131vDs5Ca).
References


18th International Conference of the System Dynamics Society. Bergen, Norway, System
Dynamics Society.
Petzold, LR. 1982. Description of DASSL: a differential/algebraic system solver. No. SAND-82-
8637; CONF-820810-21. Sandia National Labs., Livermore, CA, USA.
thesis. The University of Bergen, Norway. URL: https://bora.uib.no/bora-xm录i/
handle/1956/6154 (visited on June 17, 2022).
and Improvement Opportunities. In Proceedings of the 22nd International Conference of
//proceedings.systemdynamics.org/2004/SDS_2004/PAPERS/324SOTAQ.pdf (visited
on March 18, 2022).
Sovilj, S, Etinger, D, Sirotić, Z, Pripužić, K. 2021. System dynamics modeling and simulation
Irwin/McGraw-Hill, Boston, MA.
behaviour and policy analysis. In Proceedings of the 25th International Conference of the
System Dynamics Society and 50th Anniversary Celebration. Boston, MA, USA, System
Dynamics Society.