

Object Oriented Extensions to System Dynamics

Magne Myrtveit

Founder and Senior Vice President

Powersim AS, Hellandsneset, N-5936 Manger, Norway

Telephone: +47 88 02 34 34 – Facsimile: +47 56 37 35 00

E-mail: magne.myrtveit@powersim.no

Web page: <http://www.powersim.no>

1 Abstract

This paper describes Object Oriented (OO) extensions that can be made to the System Dynamics (SD) modeling languages. A component is a model piece that can be used as a building block of another component. As such, the component corresponds to a class in the OO world. A component is very similar to a normal variable, except that it can hold other variables and that it has a customizable interface for communicating with the rest of the model. Polymorphism is achieved through the component interfaces, as components with equal interfaces are interchangeable. This can be used to define several alternative solutions (models) to a problem, and quickly change solutions to see their effects on the whole model.

The basic SD modeling languages contain abstract building blocks (levels and non-levels) for creating models in any domain. The introduction of components makes it possible to create concrete building blocks within a specific domain. Domain specific building blocks create new and exciting opportunities for the system dynamics world, e.g. model re-use, industry specific component catalogs, quality control, standardization, and division of labor between component maker (fabrication) and component user (assembly). It can be expected that a market will develop around components, both within corporations and on the web.

Links of basic SD can be connected freely between variables. To complete a connection, the definition of the variable at the head of a link must be edited by entering a mathematical expression. The proposed extensions to SD include sockets, plugs and wires. Sockets and plugs are typed interfaces. Wires can only be connected between a matching pair of sockets and plugs, and the mathematical definition of the model will be updated automatically to reflect the connection. This technology will make it possible for non-technical users to create models by inserting ready-made components into a diagram and connecting up components using wires. Models will be runnable at any stage of the development process.

2 Overview

The paper covers high-level concepts as well as detailed descriptions of features behind Object-Oriented extensions to System Dynamics. There are also numerous examples.

Chapter 3, *Introduction*, describes the main reasons for introducing object-oriented capabilities to SD modeling languages. For readers who are mainly interested in getting an overview of concepts, features and benefits of object-orientated SD, it should be enough to read chapter 3.

Chapter 4 is about how *Equations and diagrams* represent models as text and symbols. The chapter is quite short, and aims at defining the rules for correct visualizations of models. The chapter can be skipped unless you are particularly interested in the symbolic languages for modeling.

Chapter 5 is about *Hierarchies*. The main benefit of hierarchical models is better structuring through different levels of abstraction. The main feature that is introduced here is that any variable can contain other variables. This is the way model hierarchies are built in the SD world.

The chapter also introduces our main example. The example is developed in seven complete steps through this and the following chapters. In §5.2 we start out by making a flat model into a hierarchical version. The text is quite detailed, but the equations can be skipped unless you are interested in understanding the subject down to the very detail. Many of the details will be handled automatically by software implementing these capabilities.

Chapter 6 describes *Components*. Here we define SD support for the four key characteristics of object-orientation:

objects and classes
interfaces and implementations

In the field of software engineering the use of these concepts has contributed to conceptual clarity, reduced development time, increased re-use, greater flexibility, and easier project management (evolutionary development). The reasons behind these benefits are also valid for the field of dynamic modeling:

Object architecture brings model structure closer to real-world system.
Components (classes) can be re-used within and between models.
Objects that communicate via interfaces create flexibility (polymorphism).
Interfaces, hierarchy and private substructures reduce conceptual complexity.

The main feature that is introduced here is that any model is a component. Components can be re-used in creating other models. (In OO terms, a component is a class, and a model variable is an object.)

In §6.3 our hierarchical example from §5.2 is used to create two re-usable components, a *Retailer* and a *Market*, which in turn are used to create a component based model. Section 6.4 goes one step further, in that a component is used two times inside the same model. (In OO terms, this is called “multiple instances” of a class.) Section 6.5 is about polymorphism and component swapping. This section describes how one component can take the role of another, given that they support the same interface (imported and exported variables). (In OO terms, this is called polymorphism.) Components used as functions is described in §6.6. In §6.7 we show how components can be wrapped around other types of models, such as spreadsheets and databases.

Chapter 7 about *Connections* describes an even higher level of modeling. Here, models are created by dragging components into a diagram and linking them together. The component wiring process is a guided process, where the system can hint about possible connections, and prevent many of the errors that can be done when connecting up variables at a lower level. With component wiring, there is also no need for typing in equations in order to complete a model. The main features that are introduced are the sockets and the plugs, which define type-safe interfaces for connecting up components. This chapter also describes wire flows, the object

version of the normal flow symbols of accumulator-flow diagrams. Arrays are also discussed briefly as a means of defining relationships with a multiplicity other than one. Arrays can be used to create one to many relationships, for example between a single resource and many consumers. §7.3 shows how (average) attributes can be associated with levels and managed through co-flows.

Chapter 7.3 is about *Co-flows*. Here it is described how causal-loop diagrams can be used together with accumulator-flow diagrams to visualize two different aspects of a model—the model’s topology and the feedback structure. The latter can also show the relative contribution of different substructures on overall behavior mode of the model (diverging or converging).

Chapter 9 contains *Acknowledgments*.

Chapter 10 contains *References*.

Appendix A is about measurement units, an important variable property, used extensively by the examples in this paper.

Appendix B describes the syntax used for model equations in this paper.

Appendix C contains a brief summary of the presented extensions to the basic language of SD.

Appendix D gives an overview of terminology used in this paper.

3 Introduction

If everything we built had to be done from the bottom and up, we would still live in the pre industrial age in terms of production. Imagine that one person would build an entire house, including the tools that he would need—hammer, saw, nails, glass, go to the forest and cut logs, etc. The fact that houses are built from components, and that labor is specialized into various occupations (plumber, electrician, carpenter, etc.) make houses faster and cheaper to build. Maintainability and reliability are also improved (less risk of water leakage, electrical short-circuits, and walls falling down). And, in fact, components also make it possible for the end user to make significant repairs or extensions to his house. Who has not changed a light bulb or facet himself, or installed a fan in his home?

The component technology described in this paper is about exactly the same, just in another domain—the domain of SD modeling instead of home improvement. The component technology makes it possible for specialists to create, verify and document new model building blocks that can be used in model construction. Model components behave like atomic building blocks, much like the variables of basic SD. A component can, however, be opened up for inspection and change.

Components can be made such that they can be assembled together to form a model without ever seeing a basic SD variable or entering a definition for a variable. The assembly process is a guided process, in that the modeler will be hinted about potential meaningful component connections, and the system will also prevent certain connections to take place. Compare this to connectable components that we use in a house. Such components need to be plugged into some kind of outlet before use. Standards are defined for the different interfaces, such as phone sockets and electrical outlets. It takes just a brief look to determine if a socket can be used for plugging your computer or connecting your modem. You cannot connect into the wrong socket, since the

“interfaces” do not match. In component-based modeling we use similar ideas to manifest possible connections, and prevent impossible connections.

You do not have to open up the modem with a screwdriver in order to connect in to the modem line. The necessary connections are prepared in advance inside the modem and at the other end of the modem line. When the plug goes into the socket, the wires inside the plug automatically get connected to the corresponding wires of the socket. In the component world of modeling we also have sockets and plugs, but these are now variables that can be plugged together to form connections. Inside sockets and plugs we can put individual parameters that are going to carry information once a connection is made between a plug and a matching socket. The equations behind each parameter are pre-built into the model of each component, and become effective automatically when a connection is made.

3.1 Characteristics of Basic SD

Before going into the details of component based models, let us take a brief look at basic SD. The language of basic SD has only two building blocks, the state and the flow. Yet, basic SD is a very powerful language, which can be used to express almost any dynamic system. This is due to the general nature of states and flows. States represent the system at one moment in time, and flows represent the changes to the system state when time advances.

States are represented as variables, and they are also called accumulators, levels, reservoirs and stocks. Flows are not variables, but they are normally controlled by variables. A flow expresses a rate of change, i.e., how quickly a change takes place to a state. Models can be built only from flows and states, but in practice it can be useful to introduce auxiliary variables for expressing some of the logic behind the computation of flows.

- **Flat structure is fine for small models—Hierarchy is needed for large models**

The power and beauty of basic SD lie in its simplicity and flexibility. A skilled modeler can capture the essential dynamics of a large system using only a few variables. In Forrester’s *World Dynamics* (Forrester 1973) a world model is presented. How big is the model—? It has five state variables!

There are many reasons for keeping models small. One of them is that the main dynamics of a system is normally related to a few key (state) variables. Another reason is that SD models normally do not aim at producing quantitatively accurate results. The effect of adding another variable to a model soon becomes marginal once the main structures are in place. In fact, the growing complexity of the model structure soon outweighs the benefits of any extra accuracy that can be achieved by adding more variables. (Large models are difficult to create, understand, maintain and describe to others.)

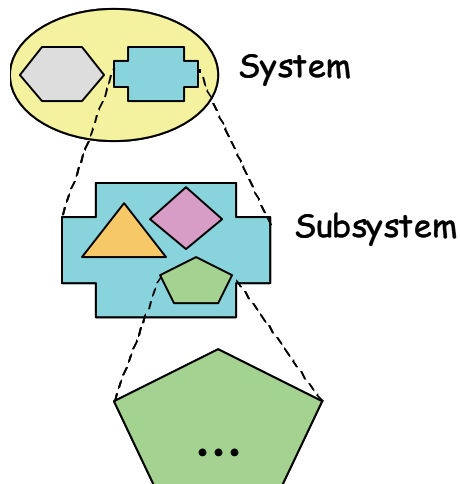


Figure 1: Systems and subsystems

There is, however, a need for dynamic modeling also in areas where detail and accuracy is of importance. Many organizations produce enormous spreadsheet models of highly dynamic business systems. The concepts of SD are better suited for modeling many of the strategic problems of the business world, and also within areas such as budgeting and forecasting. In these and similar areas models sometimes need to be large in order to capture the necessary detail.

Basic SD supports only flat models. This is acceptable for small models, but with increasing model size it becomes more and more difficult to model without a hierarchical structure. This may be one of the main reasons why SD modeling is not more widely used in day-to-day business planning.

“Structure exists in many layers or hierarchies. Within any structure there can be substructures.” (Forrester 1990, p-4-1)

In order to cope with complexity, it should be possible to introduce different levels of abstraction into a model. At the highest level, the model represents some system. At the next level we have subsystems that can be further divided into smaller and smaller parts, all the way down to the basic building blocks (accumulators and flows) of basic SD. Some software vendors use various kinds of visual filtering mechanisms (e.g., sectors) in order to deal with complexity in large models. These solutions are not as powerful as a true hierarchical solution, however.

- **Abstract concepts are fine for experts—Everybody else asks for concreteness**

Large SD models typically belong to several problem domains. Within each domain, there are domain-specific concepts. These concepts are intuitively known and understood by domain experts. These people are typically also the problem owners—the primary customers for our models.

The states and flows of basic SD are extremely general and flexible building blocks that can be used to model any dynamic system. At the same time this is a weakness, as it is in general hard to map the concrete concepts of a given problem domain to a set of abstract SD buildings blocks.

This gap makes it difficult to involve the problem owners in the modeling process, and also to explain a model to others—even if they are experts in the problem domain.

We therefore need to be able to create specific building blocks for given problem domains, functions, processes or systems.

There are some partial solutions available to this problem. Molecule models (Eberlein and Hines 1996) are pre-built model structures that can be inserted into a model. Molecules are not true building blocks, as they consist of several variables that reside on the same level as the remaining variables of the model (Tignor and Myrtveit 2000). Another approach is to increase the number of built-in variable types. Examples include ovens, conveyors and queues. The main drawback with this solution is that modelers cannot create their own building blocks. Another weakness is that specialized built-in variable types become “black box” components of a model, inaccessible for inspection and analysis.

3.2 Benefits of Object Oriented Extensions to SD

One of the main design goals for adding object-orientation to SD has been to make incremental extensions rather than fundamental changes to the concepts of basic SD. It has also been important to define the semantics of the extensions in terms of basic SD constructs.

The extensions to SD include support for true hierarchical models and for user-defined building blocks. Object Oriented SD (OOSD) being a novel approach, it is hard to estimate the extent of its potential. What we do know, is that the much related field of software engineering has greatly benefited from Object Oriented technology since its beginning in Norway in the mid 1960s, when O.J. Dahl and K. Nygaard developed Simula. Simula introduced important object-oriented concepts such as objects, classes, inheritance and dynamic binding (Birtwistle, G.M et al 1973). One would expect that some, if not all, of these benefits can be made available to the field of dynamic modeling as well. A list of some of the potential benefits follows.

- **Components enhance conceptualization and visualization of structure**

Structure is a key both to building models and to communication models to others.

“Without an organizing structure, knowledge is a mere collection of observations, practices, and conflicting incidents.” (Forrester 1990, p 1-2)

The object-oriented concept of an object can be mapped to objects of the real world, such as markets, products, distribution channels, etc. Objects are intuitive representations of real-world phenomena, and make it easier to map real-world systems to models (modeling), and vice versa (communication). Objects are hierarchical, the way the world is composed from structures and substructures. Systems composed from objects, can be displayed at various level of detail, ranging from an overall system overview down to the individual basic building blocks of the lowest level objects.

Compared to this, a normal accumulator-flow diagram of a medium size model looks like “spaghetti” to most people, and does not at all communicate the “organizing structure” of the system in a clear and meaningful way.

- **Components are easy to use, so more people can become modelers**

It is very hard for most people to create their own clip art gallery. But once a person gets access to existing pictures, he may be able to put together nice documents or presentations in very limited time. Similarly, it is much easier to put together ready-made model components than it is

to build models from scratch. Assuming that the right components are available, it is quite easy to put them together and make the necessary connections in order to make it all work.

- **Components are re-usable and can change the way models are built**

Custom made building blocks can make it much easier to re-use models or parts of models in several projects. Current solutions to this problem involve copying and pasting, and manual adjustments in order to obtain a running model. With true re-usable components, the picture becomes quite different.

One way to construct a model is to start with an empty model and insert and connect building blocks until the model is finished. This approach is called bottom-up development. With components, the amount of work can be reduced, as the modeler does not have to create all the building blocks from scratch.

Another way to construct a model is to start at the system level with a set of high-level components representing subsystems or sectors. The model is refined by customizing and refining subsystems to the degree that is necessary to capture the desired detail of the system that is being modeled. This is called top-down development. The top-down approach has similar advantages as starting with an outline when writing a paper or a presentation. Once the overall structure is in place, it is time to add details in each of the sections until the piece is finished.

From a re-use perspective, it seems that many systems are organized in similar ways when we look at the topmost levels. But once you start looking into the details of how the business is run and how decisions are made, things look differently. A top-down approach is ideal for this situation. It is conceivable that a relatively small set of high-level models can capture the most important structures of a large number of businesses. Such high-level models are called design patterns (Tignor and Myrtveit 2000) in the object-oriented world. When facing a specific modeling task, the availability of already existing design patterns can significantly reduce the time it takes to reach a finished model.

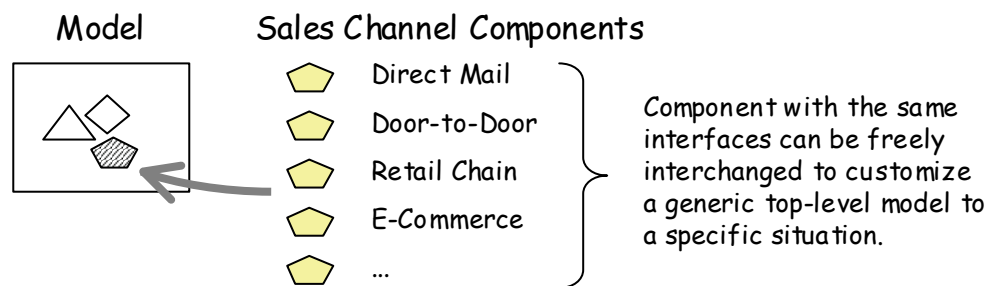


Figure 2: Interchangeable components make flexible models

The component technology described in this paper has powerful mechanisms for top-down modeling. Models can be constructed as a hierarchy of submodels. Each submodel can be implemented by any component with a matching interface. Components with similar interfaces can be interchanged freely, with no effect on the rest of the model. As an example, a model of a company may have sales channel as a high-level building block. Various sales channel implementation can be tested out by swapping in out different sales channel components. Sales channel components could be found in a component catalog, or they can be created on demand. Possible alternatives could include direct mail, chain of retailers, e-commerce, door-to-door sales, etc.

Using the object-oriented approach, it is possible to start out with a conceptual analysis of the system, identifying high-level objects and object relationships. At the next level, a system specification is created, with detailed definition of the object interfaces and connections. At the lowest level, the model is actually implemented through the use of components and/or built-in variable types.

- **Component level Quality Assurance**

The ability to create re-usable building blocks within a problem domain makes it easier to perform quality assurance. Components can be run as stand-alone models, as well as part of a larger model. Subject matter experts can validate that a given model component functions correctly, and write a component specification for use by modelers who want to use the component as part of a larger model.

- **Component Catalogs and a Market for Models**

The act of creating a set of coherent components that can be used for modeling within a certain problem domain is not trivial. It is necessary to have subject matter expertise within the problem domain. In addition, engineering skills are required in order to analyze the concepts of the problem domain and come up with an object-oriented design that identifies the components (classes and objects) that need to be made, and the interfaces of these components (how they communicate with each other). Sometimes it is straightforward to find the objects of a system and the boundaries for each object. This is normally the case when modeling physical systems, where we can make one model component for each physical component. When making models of soft systems, like markets and organizations, things become significantly harder. Finally, modeling skills are needed in order to map the characteristics of the objects into variables and equations of one of the SD modeling languages.

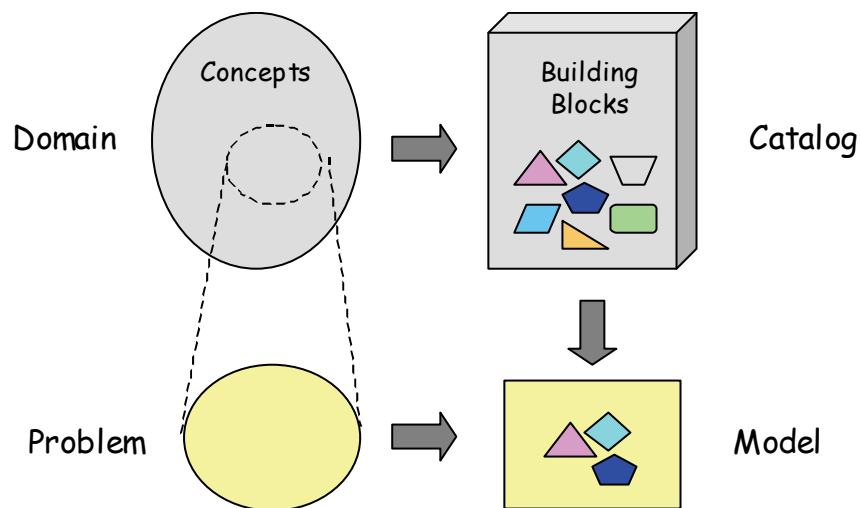


Figure 3: Component catalogs can capture domain knowledge

The challenge of creating high-quality, re-usable component catalogs opens up a possibility for branding and selling such catalogs. Only a limited number of people have the desire, skills or resources it takes to create good components. This means that there is a potential for division of labor between the component makers and the components users. The first group fabricates components, while the second (and potentially much larger group) assembles components together to make models.

- **Components extend the modeling language**

The modeling languages come with a set of primitives that can be used to create models. The primitives include functions and variable types. In normal programming languages, new functions (also called procedures and subroutines) can be created by the user. Many programming languages also support user-defined types. This is in particular the case for object-oriented languages, where types are defined as classes, and variables are objects of given classes.

A model component has similar characteristics as a class in the object-oriented world. A component has a user-defined type, defining the interface of the component, e.g., what goes in and what goes out. It is possible to create multiple instances of a component in a model. As such, a component is similar to a variable type. The same way as you can create as many instances as you want of the level (state) variable type, you can create multiple instances of a user-defined model component.

As an example, special variable types such as ovens, queues and conveyors can be built as user defined components. The advantage with components is that the new types are open for inspection and modification, and that the list of building blocks can be expanded when new needs occur.

It is also possible to use components to define new functions. Some languages support macros for this purpose. When functions get implemented as components, each function can be tested and documented like a normal component. The modeler also has the freedom to choose from time to time if a component should be invoked as a function or inserted into the model as a composite variable structure.

4 Equations and diagrams

The models in this paper will be described both as equations and as diagrams. The syntax that is used for the equations language is described in detail in Appendix B. The syntax is chosen so that it becomes easy to describe hierarchical models as well as flat models. A model is looked upon as a collection of objects that can hold other objects. Each object is defined using the following schema:

```
<type> <name> {           // Start definition of object of given <type> and with the given <name>
  <body>                   // Properties and subobjects
}
```

Below is an example of how an auxiliary variable can be defined:

```
aux Revenue {           // Start of definition of auxiliary variable Revenue
  def = Sales * Price   // Definition
  doc = "Product revenue" // Documentation
  unit = EUR            // Unit of measure
}
```

The diagramming language is an extension to accumulator-flow-diagramming. Such diagrams are also called stock-and-flow diagrams. In this paper we use the term Powersim Constructor diagram.

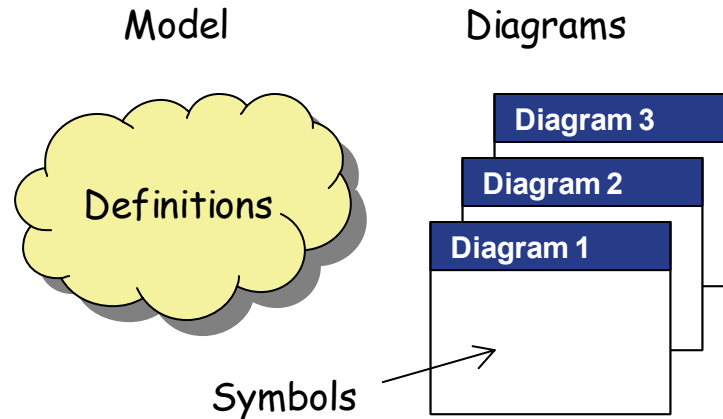


Figure 4: Multiple diagrams displaying different aspects of a model

There is a close relationship between a diagram of a model and the underlying structure, as defined by the equations. For hierarchical models it is useful to have multiple diagrams, one per submodel. Even for flat models, multiple diagrams can be useful, for example in order to focus on different aspects of the model in separate diagrams. It is important, however, that every diagram must give a picture of the model that is consistent with the underlying equations. This leads to three fundamental rules for model diagrams:

Table 1: General rules for diagrams

- 1 A diagram *needs not be a complete* representation of the underlying model, i.e., there may be variables in the underlying model that are not visualized in any particular diagram, or in any diagram at all.
- 2 Diagrams of the same model *need not be disjoint*, i.e., two or more diagrams may show the same variable.
- 3 A diagram *must be consistent* with the underlying model. (A diagram is consistent if it contains symbols that are either consistent or marked as inconsistent.)

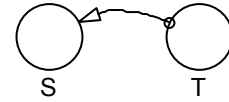
Rule number 2 implies that diagrams are independent of each other. The fact that one diagram displays a variable does not prevent another diagram from showing the same variable.

The above rules should hold for any symbolic representation of a model. For a given diagram type, we need to define what is meant by a consistent diagram. For Powersim Constructor diagrams consistency is connected to the symbol types for level, auxiliary, flow and link. The rules are like this:

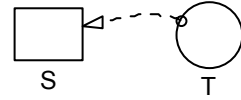
Table 2: Rules for consistent flat Powersim Constructor diagrams symbols

- 3.1 For every symbol that represents a variable, there must exist a corresponding variable in the underlying model.
- 3.2 For every variable symbol S in a diagram (excluding snapshots), the following must be fulfilled:

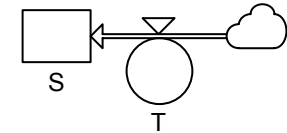
3.2.1 If S is an auxiliary symbol, and there is a symbol T in the same diagram representing a variable V(T) that the variable V(S) depends on, there must be exactly one link from T or a snapshot of T to S.



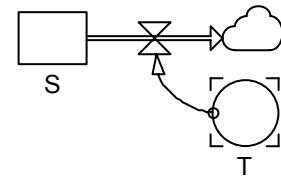
3.2.2 If S is a level symbol, and V(S) is well defined, and there is a symbol T in the same diagram representing a variable V(T) that the initialization of V(S) depends on, there must be exactly one link from T or a snapshot of T to S.



3.2.3 If S is a level symbol, for each well defined inflow of the corresponding variable V(S) that is controlled by a variable V(T) represented by a symbol T in the current diagram, there must be exactly one flow to S which flow valve is controlled by T or a snapshot of T.



3.2.4 If S is a level symbol, for each well defined outflow of the corresponding variable V(S) that depends on a variable V(T) represented by a symbol T in the current diagram, there must be exactly one flow from S which flow valve is controlled by T or a snapshot of T.



The rules for levels and flows are made extra strict in order to avoid ambiguous diagrams. Take a look at the two diagrams below, which display the same model.

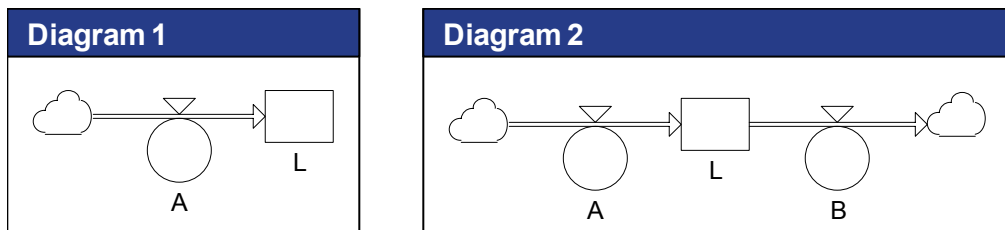


Figure 5: Two diagrams showing a level and some of its flows

How many flows has the variable L? It is quite clear that B flows out of L, but does A flow into L one or two times? To be able to answer this question precisely, we need to ensure that a variable controls no more than one inflow and one outflow of a given level. Using this rule, we can conclude that A controls only one inflow into L. (From the above diagrams, we cannot determine if there are other flows connected to L, controlled by variables not displayed in either diagram.)

When hierarchy and new variable types get introduced later in this paper, we need to extend the above rules.

Disclaimer: Powersim's implementation of object-oriented extensions to SD may depart from the description given in this paper. In particular, the graphical symbols and the syntax for equations may change.

5 Hierarchies

Purpose of solution: Dealing with complexity through multiple levels of abstraction

Core of solution: All variables can hold other variables. All variables can have diagrams.

The object-oriented extensions to SD include support for hierarchical models and re-usable components. The component concept is based on the mechanisms for hierarchy, so we will start with that.

5.1 Variables that hold other variables

It turns out that model hierarchy can be added relatively easily by allowing a variable to hold other variables. It is possible to introduce variable nesting without adding any new variable types, but it is convenient also to have a special **submodel** variable type for building model hierarchies. In the following sections we will use the new **submodel** variable type to create model hierarchies, although a similar result can be achieved using auxiliaries, for example.

The variables inside another variable are called children, and the owning variable is called the parent. Children reside one level below their parent.

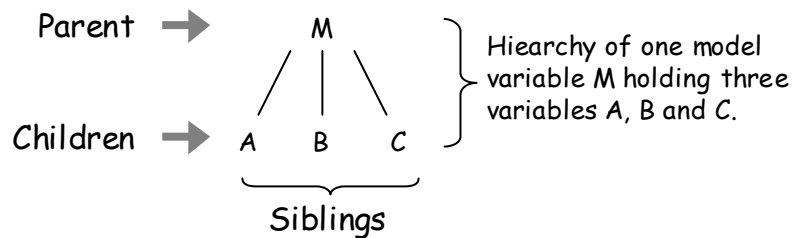


Figure 6: Parent, child, sibling relationships between variables

Children can be parents of other variables, so a variable hierarchy can be nested to any depth.

- **Information hiding is controlled through private and public variables**

An important aspect of abstraction in general, and OO in particular, is that of information hiding. Child variables can be marked as public or private. Private variables can only be accessed by their parent and by sibling variables. Public variables, on the other hand, can be accessed outside the scope of the parent variable.

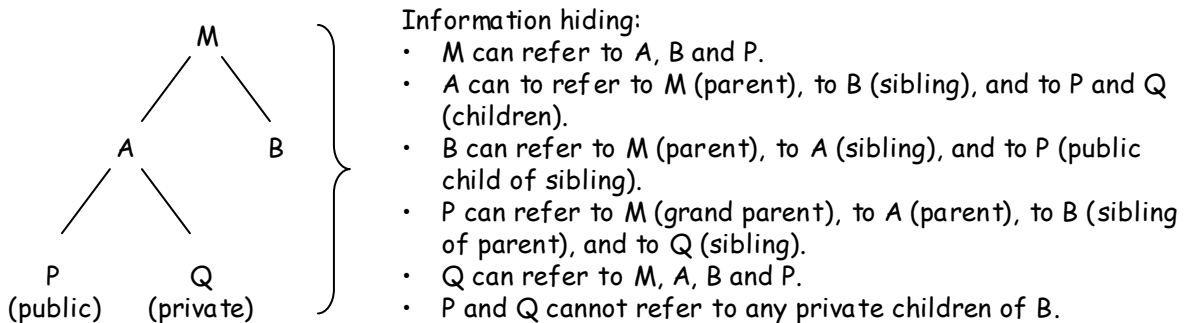


Figure 7: Encapsulation through private variables

Variables are by default marked as private.

- **Each variable has its own Powersim Constructor diagram**

For a flat model, the Powersim Constructor diagram shows all the variables of the model (or a subset thereof). The variables of a flat model can be looked upon as children of the model itself, which means that a flat model is really a hierarchy one level deep. When more levels are added to the hierarchy, we treat each parent variable as a submodel. Every variable has its own set of Powersim Constructor diagrams, displaying some or all of that variable's children.

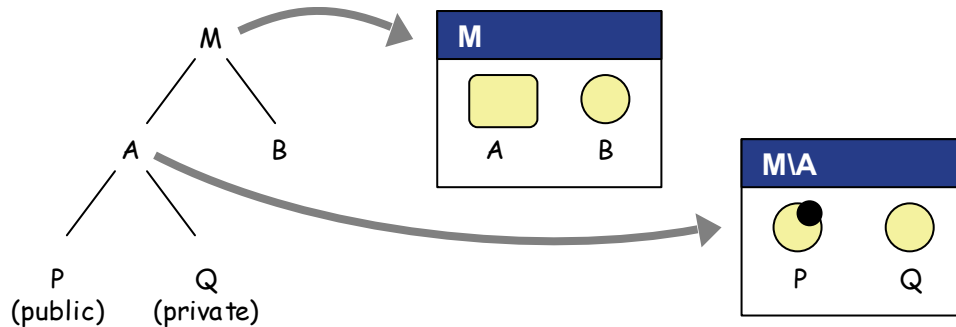


Figure 8: All variables can have diagrams

The fact that a Powersim Constructor diagram only displays children of a given parent variable is a powerful visual filtering mechanism, contributing significantly to the information hiding capabilities of OO SD.

The Powersim Constructor diagram symbol for a **submodel** variable is a rectangular shape with rounded corners. It is used for the variable A in the above example. The diagrams for P, Q and B will be empty as long as these variables have no children.

The circle at the upper, right hand corner of P inside the diagram for M/A marks P as a public variable.

- **Public variables can be displayed along with their parent symbol**

Public variables are used for carrying information up and down the hierarchy. This means that public variables should be accessible both from the outside and from the inside of the parent variable. Since the parent resides in one Powersim Constructor diagram, and the children in another, we need to be able to display public variables in both places. Children are displayed in the parent's diagram, whereas the parent is displayed inside the diagram of the children's grandparent. Public children can be visualized along with the parent symbol inside a grandparent diagram. Child symbols can be displayed as resident or as satellites. In the resident position, children are placed along the border of the parent symbols. In the satellite position, children and displayed away from the parent, and are connected to the parent by a line.

Any variable can have more than one diagram. In the figure below we have made three diagrams for M, each with a different way to display the public child of A.

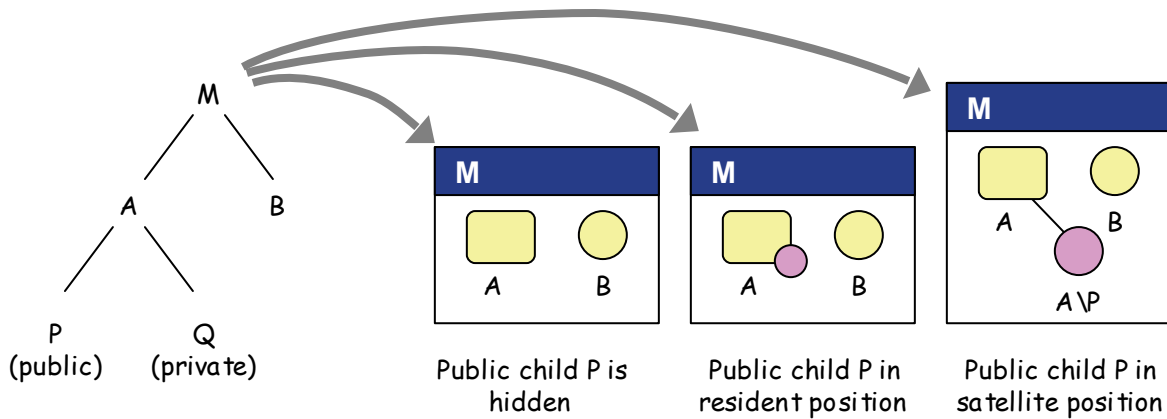


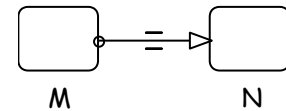
Figure 9: Public children can be hidden or displayed in different ways

Software supporting nested variables should include a command for including and excluding public children of a given variable from display.

Because of hidden, public children of a variable, we need to extend the rules for consistent Constructor diagrams in Table 2.

Table 3: Extra rule for hierarchical Constructor diagrams

3.2 Links and flows to and from a hidden public child of a
 .5 variable symbol S, are connected to S.



If we did not include rule 3.2.5, two submodels with hidden public children would always seem to be disconnected. This would be misleading if children of one submodel depended on children of the other submodel.

- **Path-like notation is used to refer to variables between scopes**

Each parent variable defines a new name space, and it is possible to use the same name in different scopes. It is, for example, possible for B to have a child called P in the above example.

Naming is done according to the following rules.

Table 4: Rules for variable path names

Rule	Example
1. The backslash character is used to separate path elements.	A\B
2. A path that starts with a name is relative to the parent of the current variable.	B\P
3. A single dot is used to start a path relative to the current variable.	.Q
4. Two dots are used to start a path relative to the grandparent of the current variable.	..M\C
5. A backslash is used to start an absolute path, i.e., a path that starts at the topmost level.	M\A\Q
6. The reserved word parent is used to refer one level up the hierarchy.	parent\parent

The figure below illustrates how you can refer from one place to another in the variable hierarchy.

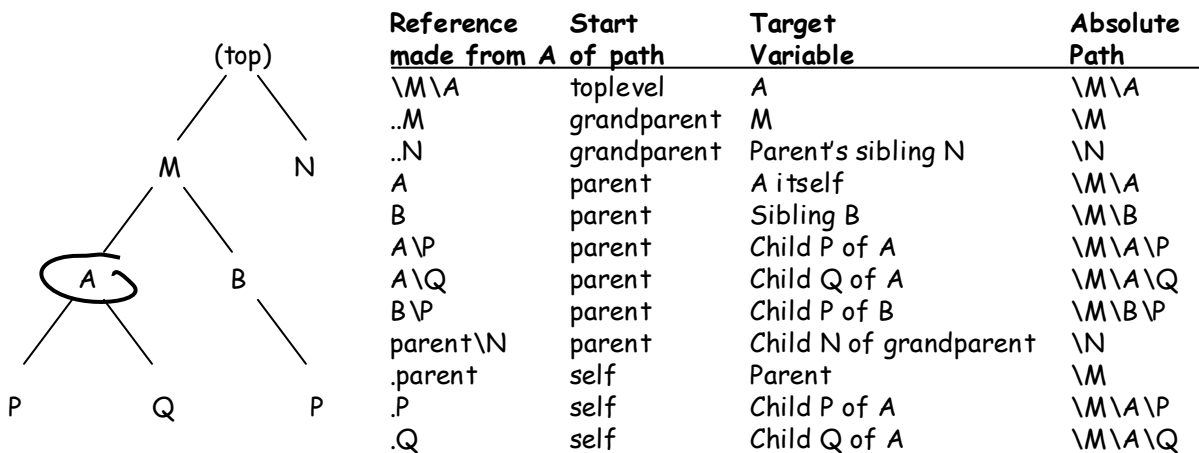


Figure 10: Path notation is used to refer to variables at different locations in the hierarchy

Observe that the notation differs from what is normally used when naming files in a hierarchical structure. The reason for this is that we want to refer to a sibling variable without qualifying its name. If the hierarchical file naming scheme was used, we would have to write `..\A + ..\B` instead of `A + B` in order to add together two variables A and B. This would clearly not be a good syntax.

- **Equations can be simplified using local variables**

Sometimes the definition of a variable gets very long, and hard to read. This is often the case when we have complex if statements or expressions with common subexpressions. In these situations it may enhance readability to introduce one or more local variables inside the target variable. Below is an example where an auxiliary variable A is defined in the normal way (left) and using a local variable T (right).

Original definition of A

```
aux A {
  def = if(C, B + C/D, 1/(B + C/D))
}
```

Modified definition of A using local variable T

```
aux A {
  aux T {
    def = ..B + ..C / ..D
  }
  def = if(C, .T, 1/.T)
}
```

- **Flows of a level can be displayed in different scopes (diagrams)**

A special situation can occur when making a level public. In this case, it will be possible to add flows to the level both from the outside and from the inside of the level's parent variable. Below is an example.

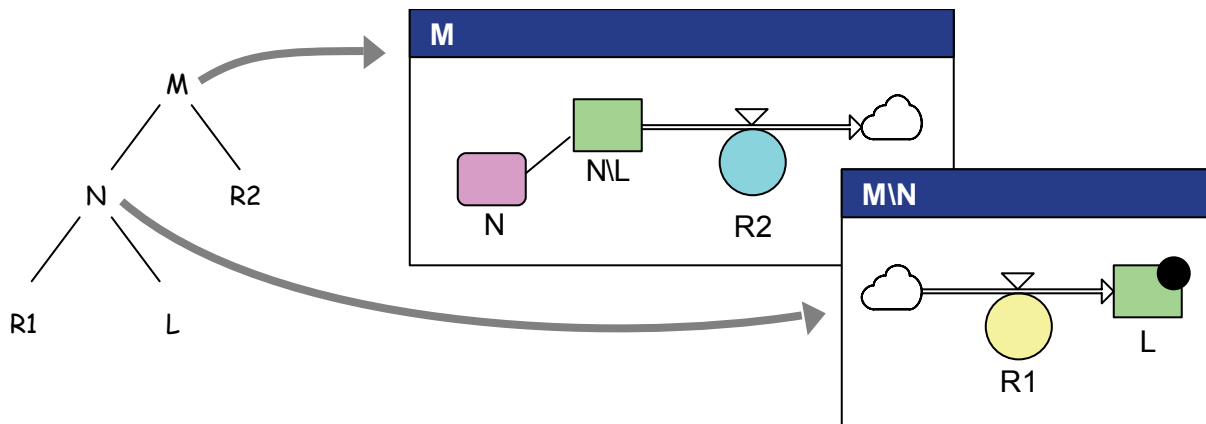


Figure 11: Flows can be attached to a level both inside and outside of its parent variable

The level variable L has one flow attached inside M/N and another one inside M. Note that the above diagrams obey the rules in Table 1 and Table 2.

The equations look like this:

```
mainmodel M {
  submodel N {
    level L {
      flows = +dt*R1 -dt*..R2
    }
    aux R1 {...}
  }
  aux R2 {...}
}
```

- **Reference definitions are used to share variables**

When one submodel passes information to another submodel, we get a situation where the same variable should actually be part of the public section of two submodels. Below is a diagram of a simple, flat model that we want to convert into a model with two submodels. Information is passed from the first submodel to the second submodel via the variable P.

There are three solutions to this problem.

1. We can leave the variable P outside both submodels. This has the disadvantage that the parent needs to hold a separate variable just for passing information on behalf of its child models. It also has the disadvantage that the second submodel needs to know about a variable in the scope of its parent.
2. We can place the variable P inside one of the submodels. The disadvantage here is that then the submodels need to know about one another. This violates the principles of information hiding.
3. We can put one copy of P in each of the submodels. This solution requires that we copy the value of the first submodel's P to the second submodel's P during simulation.

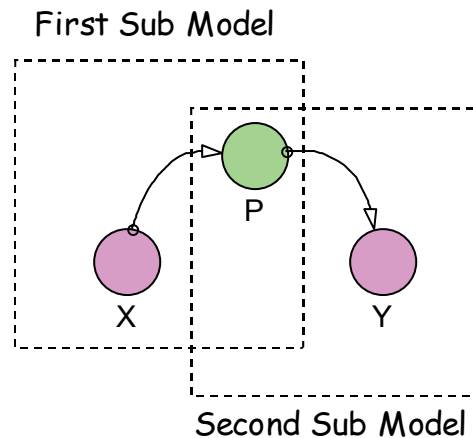


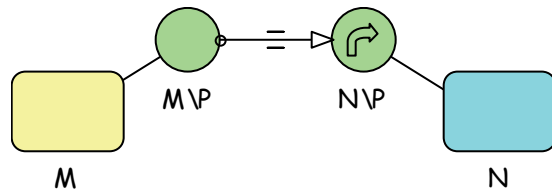
Figure 12: Situation where two submodels want to own the same variable

The third solution is clearly the best from an object-oriented perspective. If P is an auxiliary, the copying of P can be done straight forward through a normal assignment. But, in general, P can also be a level with flows attached to it. Some flows can be inside the first submodel, and others in the second. In this case, assignment cannot be used. We need a solution where the P in the first submodel and the P in the second actually share the same memory. This can be done through the concept of a reference definition.

A reference definition says that one variable shares the value with another variable. (References are similar to pointers in programming languages, and references in C++.) A variable defined as a reference variable does not have its own memory for storing values. Instead, the variable has a direct reference to the target variable's memory.

Below is an example, where two models M and N both have a parameter P sharing the same value. The value of P is defined inside M and used by N, so we define $N \setminus P$ as a reference to $M \setminus P$.

Diagram



A reference link is displayed with two short lines parallel to the link, as shown above. The link and the short lines together indicate identity (\equiv).

Equations

```

submodel M {
  aux X {...}
  aux P {
    public
    def = X
  }
}
submodel N {
  aux P {
    public
    ref = ..M\VP
  }
  aux Y {
    def = P
  }
}

```

If a variable is a reference it cannot have its own definition in addition. Therefore the **ref** property excludes the **def** property, and vice versa. A variable that is defined as a reference, is marked with the reference symbol ↻.

- **Flows can be connected to submodels**

A submodel can be used to implement a special variable type like a conveyor. For such submodels, we want the Powersim Constructor diagram to indicate that there is a flow entering the conveyor at one side, and leaving it on the other side.

It is also important to ensure that flows do not alter their direction on the outside of a submodel without also changing the inside. Below is an example where the intention is that there should be a flow from *Src* to *Dest*, but because of a wrong direction on the flow arrow connected to *Src*, the result is a model where both *Src* and *Dest* actually get filled.

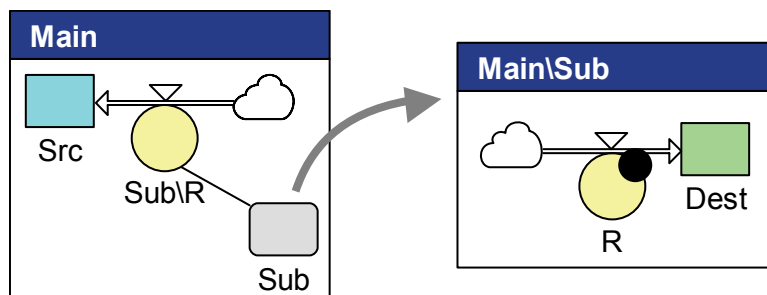


Figure 13: Parameter (R) controlling flow that is not connected to submodel (Sub)

The above model is syntactically correct, and there are situations where the same variable should control more than one inflow. Our aim is that the submodel specifies the direction of any flows that enter or leave the submodel, reducing the possibility of erroneous connections when the submodel parameters are connected up to the enclosing system.

A solution is to add the possibility of defining a parameter as an inflow or an outflow. Variables that are marked this way always get displayed with a flow symbol attached. Inside the diagram of the parent variable, inflow variables have a cloud symbol with a level inside at the tail end. The level indicates that the flow comes from a level that lies outside the submodel. It will not be

possible to attach the tail of the flow to a level inside the submodel. On the outside of the submodel, the inflow parameter can be displayed either in satellite or resident positions. The attached flow symbol will have a normal tail end, but the flow arrow will be permanently connected to the parameter's parent symbol. The flow symbol will replace the satellite line for inflow and outflow parameters. For outflows, the opposite will hold. Below is an example.

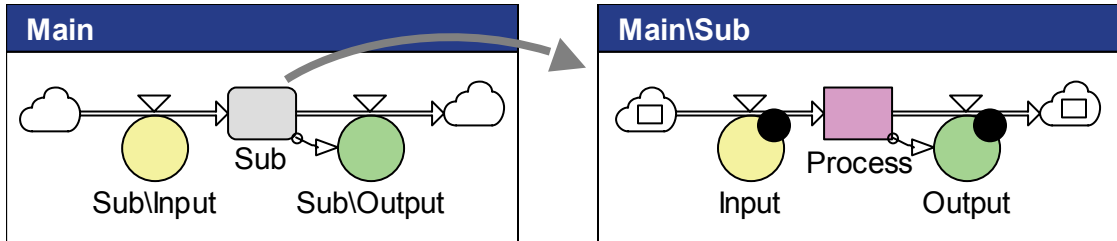


Figure 14: Inflows and outflows of a submodel

Here *Input* is marked as an inflow and *Output* as an outflow. It is not possible to connect the tail end of *Input* to a level inside *Sub*. Similarly, the head end of *Output* cannot be connected to a level. Inside *Main*, *Sub\Input* will be permanently connected to *Sub* at the head end, and *Sub\Output* will be attached at the tail. It will be possible to connect the tail of *Sub\Input* to a level. In the case of *Sub\Output*, it is the head of the arrow that can be connected to a level.

The inclusion of inflow and outflow attributes for parameters enables us to ensure consistent use of flows on the inside and on the outside of a submodel. We also achieve that flows into and out of submodels automatically get connected to the submodel symbols in the appropriate way.

- **Example of a conveyor belt submodel**

The flat model below shows a transport from a remote inventory to a local inventory.

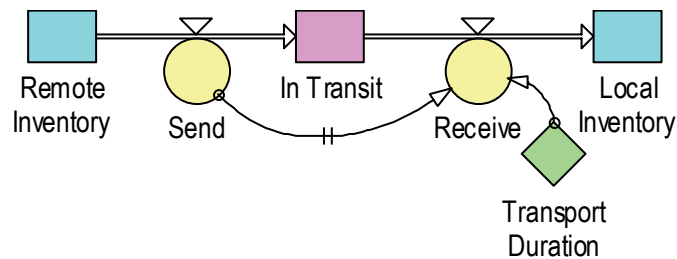


Figure 15: Flat version of transport model

As a first step, the transport can be modeled using a generic conveyer submodel, like this:

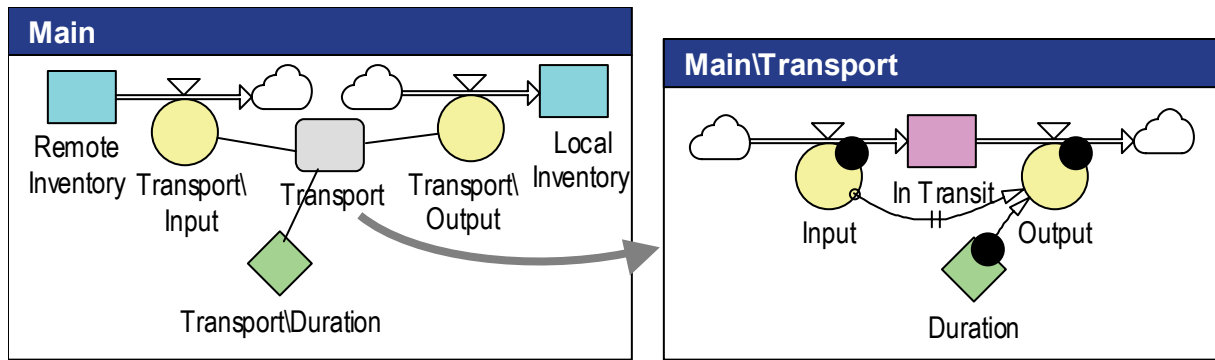


Figure 16: Hierarchical transport model with flows that end in cloud symbols

The model has been changed to include a submodel *Transport* with three public variables. The diagram of the main model does not communicate the fact that there is a flow through the *Transport* submodel. There is also no control on the use of *Transport\Input* and *Transport\Output* inside the main model.

As a final step, we therefore make *Input* an inflow and *Output* an outflow. The diagrams now become like this:

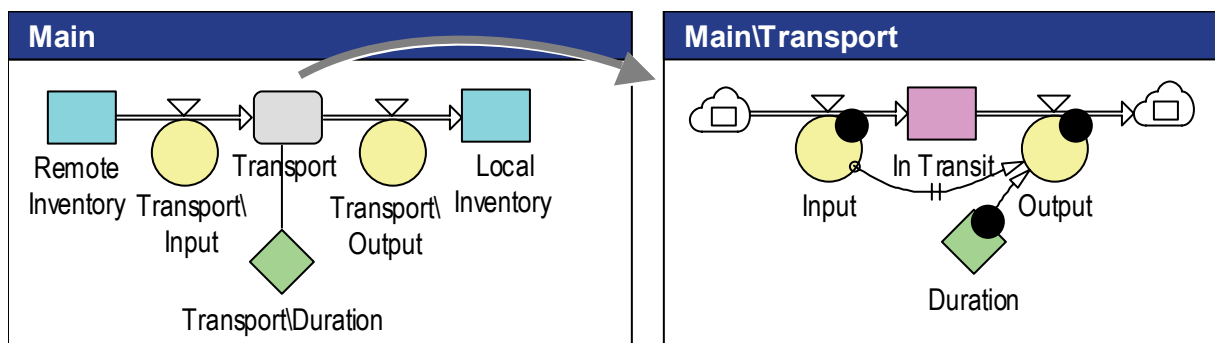


Figure 17: Hierarchical transport model with flows connected to submodel (Transport)

In the above example, the flows are labeled using path notations inside the main diagram. If you want to change the names of the flows and get rid of the path naming, you can create extra flow variables and construct cascaded flows, as described next.

- **Flows can be chained**

Let us take a look at a situation where a flow goes from one submodel into another. In this case the first submodel will have an outflow parameter, and the second submodel an inflow. How do we connect the outflow of one submodel to the inflow of the next?

The solution that we chose is to allow flows to be chained together. The cloud symbol at the head of one flow can be dropped on the cloud at the tail of another flow. The variables that are connected to different valves of the same flow must be identical. This is enforced by allowing only one flow variable of a flow chain to be defined using a normal definition. The other flow variables on each side of the controlling flow variable must be part of a reference chain that ends at the controlling variable.

Below is an example where we have two transport submodels, one feeding the other.

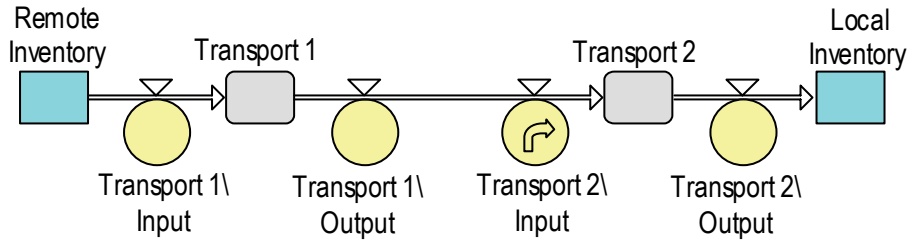


Figure 18: The outflow of one submodel is chained to an inflow of another submodel

It is possible to put more flow variables into the chain, and hide one or more parameters of the submodels *Transport 1* and *Transport 2*. Here is an example where a *Send* variable is put in before *Transport 1\ Input* and a *Receive* variable is added before the *Local Inventory*. All parameters are hidden.

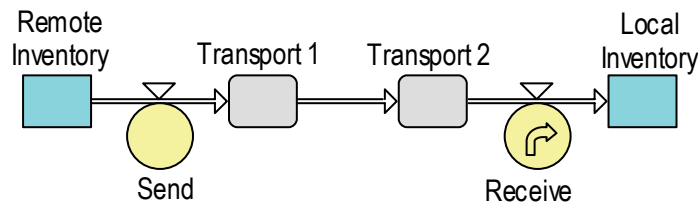


Figure 19: Chained flows between hidden parameters

5.2 Hierarchical Model Example

In this paragraph we will illustrate the use of model hierarchy by making modifications to a simple example model. The example is about supplying products to a market. The market sector is totally driven by word-of-mouth, and there is no competition. A retailer receives products from a supplier, and ships the products to the market.

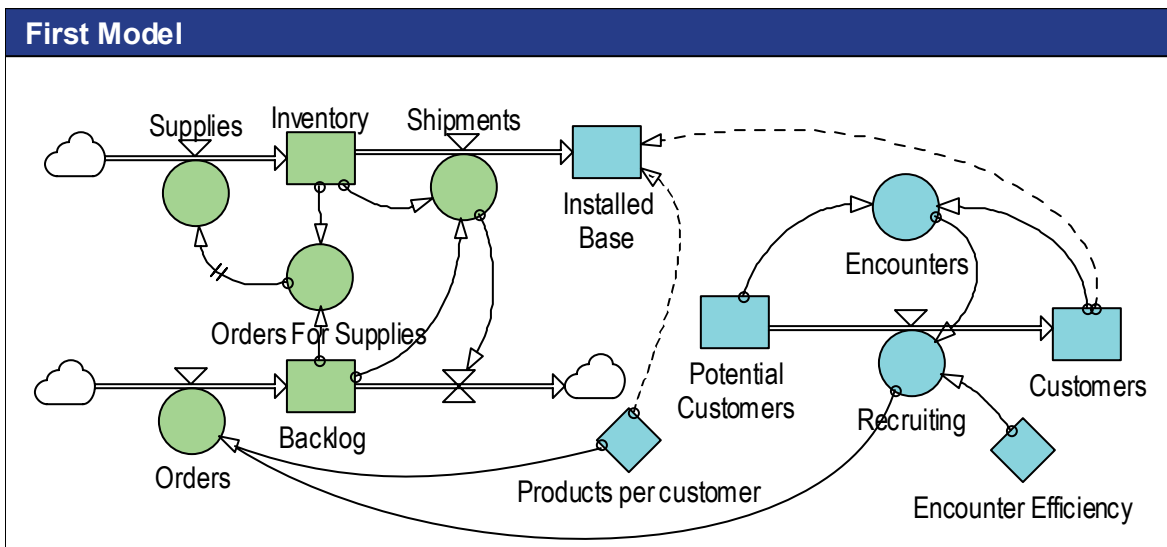


Figure 20: First Model

The objective of the retailer is to fulfill customer *Orders* by making product *Shipments*. In order to be able to deliver promptly, an *Inventory* of products is maintained. When the inventory falls

below the desired level, the retailer makes *Orders For Supplies*, which in turn leads to *Supplies* of new products.

The models in this paper make use the following measurement units. (See *Appendix A* for a description of units.)

```
unit Day {
    doc = "Time measured in days"
    note = "Predefined unit"
}
unit Encounter {
    def = atomic*)
    doc = "Encounter between customer and non-customer"
}
unit Person {
    def = atomic
}
unit Product {
    def = atomic
}
```

*) The keyword **atomic** is used to define a new unit, which is not compatible with any other unit.

The equations for the *First Model* are displayed next.

```
mainmodel 'First Model' {
    level Backlog {
        init = 0 <<Product>>
        flows = -dt*Shipments +dt*Orders
        doc = "Orders not yet fulfilled."
        unit = Product
    }
    level Customers {
        init = 1 <<Person>>
        flows = -dt*Recruiting
        doc = "Number of actual customers."
        unit = Person
    }
    aux 'Encounter Efficiency' {
        unit = Product/Encounter
        init = 0.1 <<Person/Encounter>>
        doc = "Recruited customers per encounter"
    }
    aux Encounters {
        unit = Person/Day
        def = Customers * 'Potential Customers' / (Customers + 'Potential Customers')
        doc = "Number of encounters between customer and non-customer in a day."
        unit = Encounter/Day
    }
    level 'Installed Base' {
        init = Customers * 'Products per customer'
        flows = +dt*Shipments
        doc = "Number of products in the market."
        unit = Product
    }
}
```

```

level Inventory {
    init = 0 <<Product>>
    flows = -dt*Shipments +dt*Supplies
    doc = "Products ready to ship to customers."
    unit = Product
}
aux Orders {
    def = Recruiting * 'Products per Customer'
    doc = "Product orders. Each customer buys one product"
    unit = Product/Day
}
aux 'Orders For Supplies' {
    def = MAX(0 <<Product/Day>>, Backlog - Inventory)
    doc = "Orders for more supplies."
    unit Product/Day
}
level 'Potential Customers' {
    init = 1000 <<Person>>
    flows = -dt*Recruiting
    doc "Number of potential customers."
    unit = Person
}
aux 'Products per Customer' {
    init = 1<<Product/Person>>
    doc = "Average number of products per customer."
    unit Product/ Person
}
aux Recruiting {
    def = Encounters * 'Encounter Efficiency'
    doc = "Customer orders."
    unit = Person/Day
}
aux Shipments {
    def = MIN(Inventory, Backlog)
    doc = "Product shipments to customers."
    unit = Product/Day
}
aux Supplies {
    def = DELAYMTR('Orders For Supples', STARTTIME + 4 <<Day>>)
    doc = "Received products from supplier."
    unit = Product/Day
}
}

```

In the diagram in Figure 20 the retailer and the market are implemented in one diagram. This makes it difficult to isolate one from the other. Let us make the retailer a submodel of the overall model. Software support for submodels should provide an Implode command that will convert a variable selection into a submodel in one step. The same effect can be achieved manually by following these steps.

1. Create a new submodel variable called *Retailer* inside our model.
2. Open up the diagram for the *Retailer* submodel (using *Retailer's* context menu).

3. Drag the variables that belong to the retailer from the main diagram into *Retailer's* diagram.

Step 3 will make the following changes implicitly:

4. *Orders* is made a public variable, since its definition uses information from the outside (*Recruiting*).
5. The definition of *Orders* is adjusted to reflect the changed scope of the used variable *Recruiting*.
6. *Shipments* is made a public outflow variable, since it is providing information (a flow) to the outside (Installed Base).

The revised model looks like this:

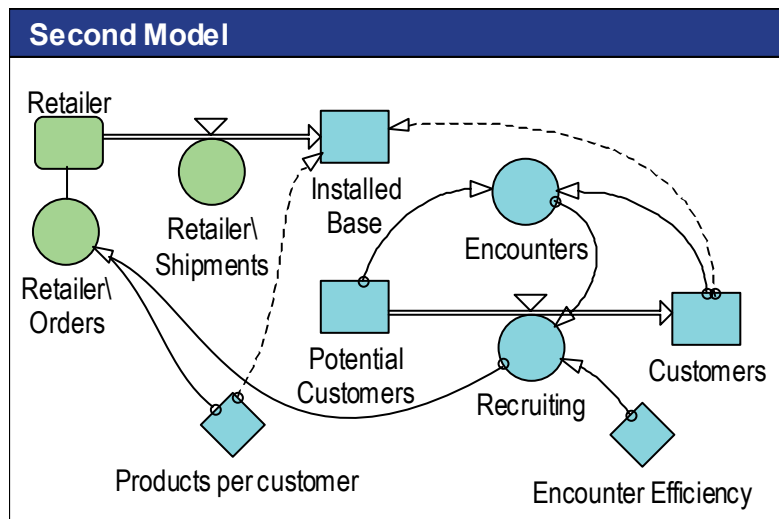


Figure 21: Second Model

Note how *Retailer\Orders* is connected to the *Retailer* submodel using a satellite line, while *Retailer\Shipments* is connected by a flow (since *Shipments* is an outflow of *Retailer*). The diagram for *Retailer* is displayed next.

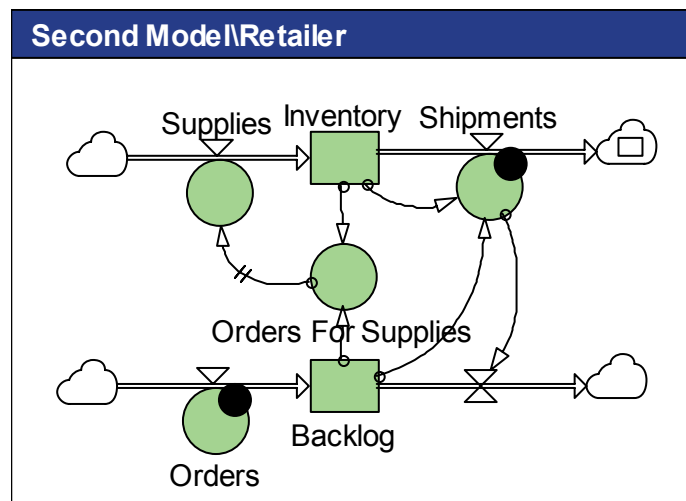


Figure 22: Retailer Submodel

Observe how the public variables *Orders* and *Shipments* are marked using filled circles.

The equations for the *Second Model* are displayed next. The retailer related variables are located inside the *Retailer* variable. Note also the use of **public** on *Orders* and *Shipments*, and **outflow** on *Shipments*. (Modified lines are boxed.)

```

mainmodel 'Second Model' {
  submodel Retailer {
    level Backlog {
      init = 0 <<Product>>
      flows = -dt*Shipments +dt*Orders
      doc = "Orders not yet fulfilled."
      unit = Product
    }
    level Inventory {
      init = 0 <<Product>>
      flows = -dt*Shipments +dt*Supplies
      doc = "Products ready to ship to customers."
      unit = Product
    }
    aux Orders {
      public
      def = ..Recruiting * 'Products per Customer'
      doc = "Product orders"
      unit = Product/Day
    }
    aux 'Orders For Supplies' {
      def = MAX(0 <<Product/Day>>, Backlog - Inventory)
      doc = "Orders for more supplies."
      unit Product/Day
    }
    aux Shipments {
      public
      outflow
      def = MIN(Inventory, Backlog)
      doc = "Product shipments to customers."
      unit = Product/Day
    }
    aux Supplies {
      def = DELAYMTR('Orders For Supplies', STARTTIME + 4 <<Day>>)
      doc = "Received products from supplier."
      unit = Product/Day
    }
  }
}

level Customers {
  init = 1 <<Person>>
  flows = -dt*Recruiting
  doc = "Number of actual customers."
  unit = Person
}
aux 'Encounter Efficiency' {
  unit = Person/Encounter
  init = 0.1 <<Person/Encounter>>
  doc = "Recruited customers per encounter"
}

```

```

aux Encounters {
    unit = Person/Day
    def = Customers * 'Potential Customers'/(Customers + 'Potential Customers')
    doc = "Number of encounters between customer and non-customer in a day."
    unit = Encounter/Day
}
level 'Installed Base' {
    init = Customers * 'Products per customer'
    flows = +dt*Retailer\Shipments
    doc = "Number of products in the market."
    unit = Product
}
level 'Potential Customers' {
    init = 1000 <<Person>>
    flows = -dt*Recruiting
    doc "Number of potential customers."
    unit = Person
}
aux 'Products per Customer' {
    init = 1<<Product/Person>>
    doc = "Average number of products per customer."
    unit Product/ Person
}
aux Recruiting {
    def = Encounters*'Encounter Efficiency'
    doc = "Customer orders."
    unit = Person/Day
}
}

```

The only changes to the equations, are that “..” is put in front of *Recruiting* inside the definition of *Orders*, and “Retailer\” is put in front of *Shipments* in the flow definition of *Installed Base*.

As a natural next step, we can put the market sector into another submodel. We can do this manually, or simply by selecting the variables of the market sector and executing the *Implode* command. It seems natural to make *Orders* and *Shipments* part of the market as well as the *Retailer*. Since *Retailer\Orders* is actually the same as *Market\Orders*, we define the former as a reference to the latter. We do the same for *Retailer\Shipments* and *Market\Shipments*. In order to make use consistent naming, *Shipments* of the *Market* is renamed to *Supplies*, and *Orders* is renamed *Orders for Supplies*.

The resulting model looks like this:

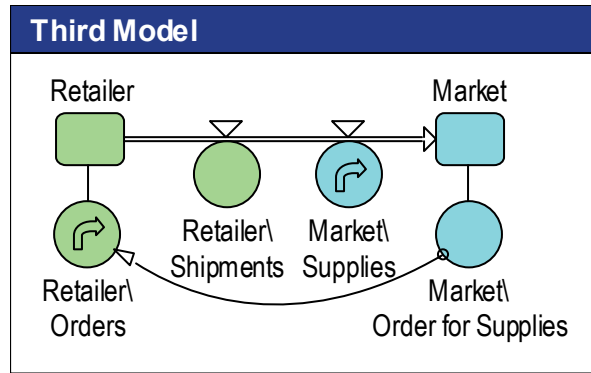


Figure 23: Third Model

The diagram for the *Market* submodel is given next.

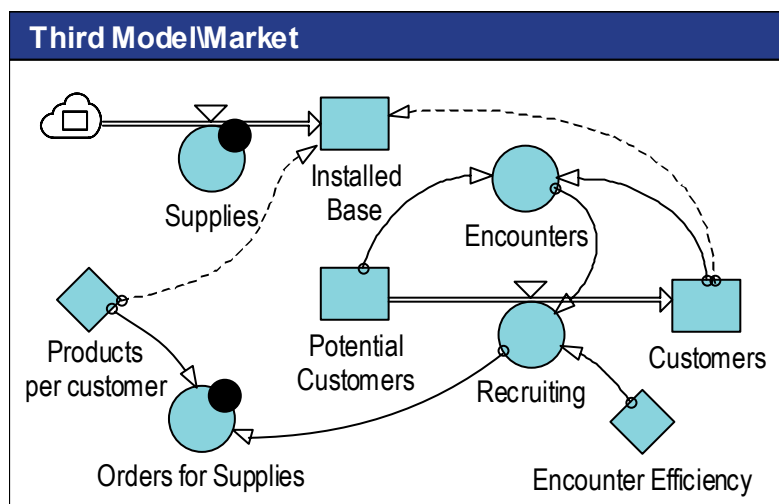


Figure 24: Market Submodel

Note the circles that mark *Orders for Supplies* and *Supplies* as public. The special cloud symbol on *Supplies*' flow indicates that *Supplies* is an inflow to the *Market*.

The top-level model in Figure 23 gives a clear picture of the structure of the system. By hiding the parameters, we can make things even simpler, like this:

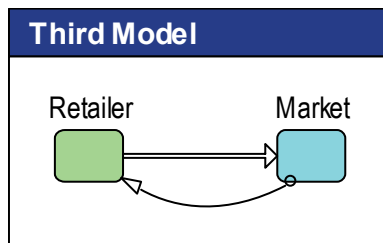


Figure 25: Third Model with hidden parameters

Links that are connected to a parameter will be connected to the parent variable when the parameter is hidden.

The equations for the *Third Model* are given next. Changed parts are put inside boxes.

```
mainmodel 'Third Model' {
```

```
  submodel Retailer {
```

```
    level Backlog {
```

```
      init = 0 <<Product>>
```

```
      flows = -dt*Shipments +dt*Orders
```

```
      doc = "Orders not yet fulfilled."
```

```
      unit = Product
```

```
    }
```

```
    level Inventory {
```

```
      init = 0 <<Product>>
```

```
      flows = -dt*Shipments +dt*Supplies
```

```
      doc = "Products ready to ship to customers."
```

```
      unit = Product
```

```
    }
```

```
    aux Orders {
```

```
      public
```

```
      ref = ..Market\Orders for Supplies'
```

```
      doc = "Orders from customers"
```

```
      unit = Product/Day
```

```
    }
```

```
    aux 'Orders For Supplies' {
```

```
      def = MAX(0 <<Product/Day>>, Backlog - Inventory)
```

```
      doc = "Orders for more supplies."
```

```
      unit Product/Day
```

```
    }
```

```
    aux Shipments {
```

```
      public
```

```
      outflow
```

```
      def = MIN(Inventory, Backlog)
```

```
      doc = "Product shipments to customers."
```

```
      unit = Product/Day
```

```
    }
```

```
    aux Supplies {
```

```
      def = DELAYMTR('Orders For Supplies', STARTTIME + 4 <<Day>>)
```

```
      doc = "Received products from supplier."
```

```
      unit = Product/Day
```

```
    }
```

```
  }
```

```
  submodel Market {
```

```
    level Customers {
```

```
      init = 1 <<Person>>
```

```
      flows = -dt*Recruiting
```

```
      doc = "Number of actual customers."
```

```
      unit = Person
```

```
    }
```

```
    aux 'Encounter Efficiency' {
```

```
      unit = Person/Encounter
```

```
      init = 0.1 <<Person/Encounter>>
```

```
      doc = "Recruited customers per encounter"
```

```
    }
```

```

aux Encounters {
    unit = Person/Day
    def = Customers * 'Potential Customers'/(Customers + 'Potential Customers')
    doc = "Number of encounters between customer and non-customer in a day."
    unit = Person/Day
}
level 'Installed Base' {
    init = Customers * 'Products per customer'
    flows = +dt*Supplies
    doc = "Number of products in the market."
    unit = Product
}
aux 'Orders for Supplies' {
    public
    def = Recruiting * 'Products per Customer'
    doc = "Product orders"
    unit = Product/Day
}
level 'Potential Customers' {
    init = 1000 <<Person>>
    flows = -dt*Recruiting
    doc "Number of potential customers."
    unit = Person
}
aux 'Products per Customer' {
    init = 1<<Product/Person>>
    doc = "Average number of products per customer."
    unit Product/Person
}
aux Recruiting {
    def = Encounters*'Encounter Efficiency'
    doc = "Customer orders."
    unit = Person/Day
}
aux Supplies {
    public
    inflow
    ref = ..Retailer\Shipments
    doc = "Products received."
    unit = Product/Day
}
}

```

The variable hierarchy for the Third Model looks like this:

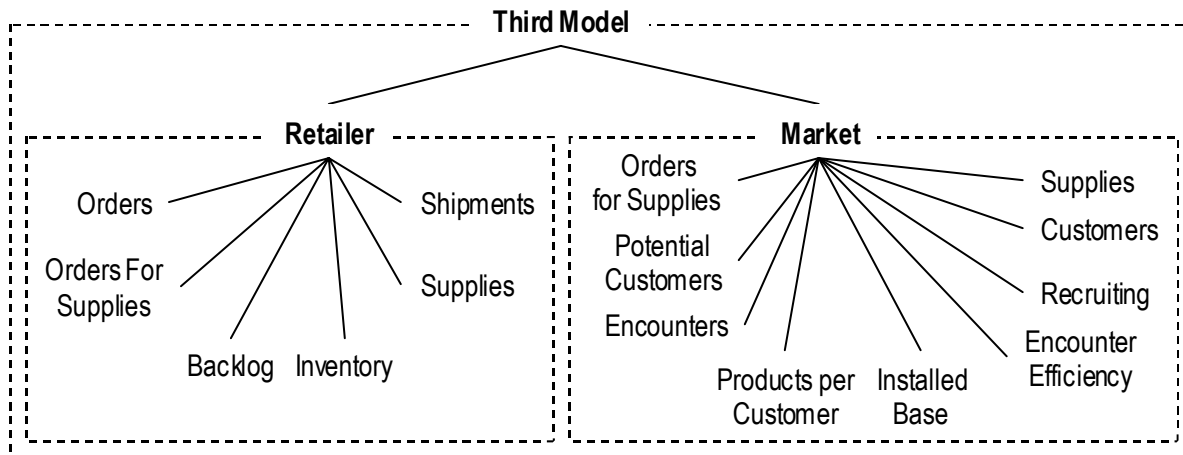


Figure 26: Variable hierarchy of Third Model

6 Components

Purpose of solution: Conceptual concreteness through objects. Coping with complexity through encapsulation. Re-use. Flexibility. Maintainability.

Core of solution: Simulation models are components that can be re-used in creating other simulation models. Division between specification (interface) and implementation (class) of objects. Interchangeable implementations of objects.

Object-oriented concepts: class, object, interface/type, implementation, message, method, event, polymorphism

The introduction of hierarchical models contributes significantly to abstraction through information hiding. But hierarchy does not give direct support for re-usable objects and polymorphism, two of the core characteristics of object-orientation.

6.1 Background

So long as there are only abstract data types and no concrete ones, there is nothing to play the role that nature plays for physical science, or that catalogs of standard raw materials play for the engineer. So long as everything we encounter in software is truly new, how can we possibly hope to benefit from anything the manufacturing age might have to offer?

Cox 1996, p. 67.

How can OO capabilities be added to System Dynamics? To find a good answer to this question, it is important to understand the difference between object-oriented concepts and object-oriented techniques. The concepts define the nature of object-orientation, and can be implemented in many different ways. Languages such as Simula, Smalltalk, Object Pascal, C++, and Java use different approaches to object-orientation. When we choose an approach for SD modeling, it is important to do this within the spirit of SD.

It is a design objective to keep the number of extensions to basic SD as small as possible. We don't want to end up with an elaborate object-oriented system where

SD in the traditional sense only remains as a small part. SD the way we are used to it should still be at the heart of the extended language.

- **In SD models, there are no sequence of operation, other than the advancing of time**

SD is very different from procedural programming languages. There is no sequence in a SD model. SD models define influences that work concurrently, and take effect on all object states simultaneously when time advances. This is very different from a sequential program, where statements are executed one at the time, updating object states along the way. In sequential and parallel processes it makes sense to pass events between objects. In SD influences are active all the time, and change is done only through accumulation or draining processes that take place through flows connected to levels. Therefore, the terms event, message and method makes little sense in the context of SD models. In a discrete event simulator, however, these techniques make perfectly good sense.

- **SD models have two built-in messages; “time change” and “value changed”**

There are only two “messages” in a traditional SD simulation (Tignor and Myrtveit 2000). Both the messages and their responses are built into the run-time system for running simulations. The first message type goes to all variables when time is advanced. The second message type goes to dependent variables, when a variable gets a new value. Only level variables perform an action in response to the “time advanced” message, adjusting their states based on the values of associated flows. This causes levels to get new values, which is an “event” that is sent to dependent variables. A non-level responds to “value changed” by re-evaluating its equation, updating its value, and sending a value changed message to its own dependants. This is one way to describe how the variables of a model get updated (from a conceptual point of view).

- **SD models work on aggregated values instead of individual items**

In SD models, we normally do not study individual objects. Instead, we work with masses of objects and their average attributes. This approach means that SD models operate on a higher level of aggregation than most discrete event models. SD models can typically be made significantly smaller than their discrete counterparts, but the aggregation of items into groups, takes away the object-oriented message passing metaphor used by most object-oriented systems.

- **Inheritance**

It can also be useful to realize that inheritance is a technique, and not a core part of object-orientation. Inheritance is primarily a way to share code between different object implementations. One of the disadvantages of inheritance is that it adds conceptual complexity and leads into difficult areas such as multiple inheritance.

6.2 The SD counterparts of objects, classes, interfaces and implementation

Object-oriented technologies depend heavily on the separation between interface and implementation. An interface describes a protocol without implementing it. The implementation of an interface is often called a class. Interfaces are sometimes also called types or abstract classes. Interfaces, types and classes are meta objects that describe real objects. Real objects are instances of classes, and all the object instances of a class share the same implementation.

If we let objects communicate via interfaces without making use of the way a particular object class implements an interface, it becomes easy to exchange one object with another object that

supports the same interface. This is called polymorphism, and can be used to create very flexible systems.

In the SD world objects are variables. The built-in variable types are classes that implement the basic functionality of levels and auxiliaries. Let us extend the functionality of submodels to include an interface definition and an implementation. The interface will define a set of variables, and the way information is transferred to and from the submodel. We use components to implement submodel interfaces. This means that any component becomes a class in the object-oriented world. The component implements an interface that can be used to access the component, or any compatible component from another model.

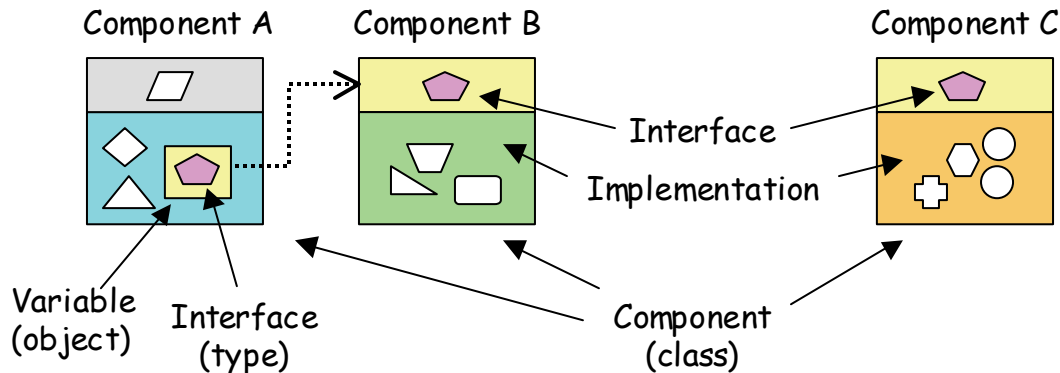


Figure 27: Components can be used as submodels of other components

The above figure illustrates a component A that has a submodel variable with an interface that is implemented in two different ways by component B and component C. The submodel of A can be connected to either of the components B or C, since their interfaces are the same. In the figure, B's implementation is used.

User-defined components and variables of the type **submodel** play together in a way that allows a submodel variable to be implemented by a component. Each time a component is used to implement a submodel, a new instance of the component is created. Any component can also be simulated as a stand-alone model. In this case a modeling environment such as POWERSIM® STUDIO creates and runs an implicit instance of the component.

This means that a **component** is the SD counterpart of the OO **class**.
 Components can be used to create instances of SD **variables**, the counterparts of OO **objects**.

- **Variable import and export**

Each variable that is part of the interface can be imported or exported by the component. It is also possible to create certain combinations of import and export, where the initial value is transferred one way and the simulated results the opposite way. This can be used to initialize a level from one side of the interface, and receive the simulated results back.

When a submodel is implemented by a component, both the submodel and the component contain an interface section. Each variable of the submodel interface is matched with a variable of the component. During simulation values are transferred between the two sides according to

the import/export settings. We call the interface variables of the component for formal parameters, and the interface variables of the submodel actual parameters.

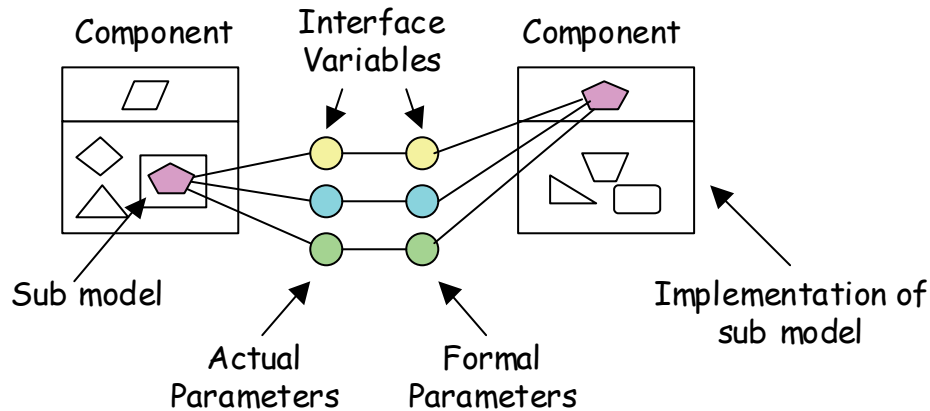


Figure 28: Components and submodels communicate via interfaces

The transfer settings of an interface are defined from the formal parameter's point of view. Full import, for example, means that the formal parameter gets its value from the actual parameter throughout the simulation.

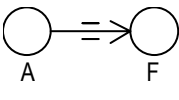
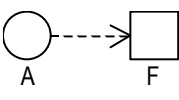
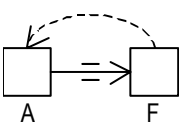
A parameter can be set to one of the following transfers.

Table 5: Variable import and export settings

Type	Symbol	Explanation
Full Import		Full import is used to import a value to a component or submodel. The formal parameter becomes a reference to the actual parameter. If the formal parameter is a level, the actual parameter must be a level as well.
Init Import		When only the initial value is to be imported, init import can be used. The formal parameter keeps its own memory, and copies the initial value from the actual parameter. The formal parameter cannot be an auxiliary.
Tail Import, Init Export		This is a special two-way transfer, which is most useful for levels. The formal parameter becomes a reference to the actual parameter, but in addition to that the actual parameter gets its initial value from the formal parameter. The actual parameter cannot be an auxiliary
Full Export		This setting has the same effect as Full Import, only that the roles of the formal and the actual parameters are reversed.
Init Export		This setting has the same effect as Init Import, only that the roles of the formal and the actual parameters are reversed.
Tail Export, Init Import		This setting has the same effect as Tail Import, only that the roles of the formal and the actual parameters are reversed.

The semantics of the various parameter transfer settings can be modeled. Below is one example of each of the import transfers, assuming that the actual parameter is A and the formal parameter is F. (The export transfers can be found by reversing the roles of A and F.)

Table 6: Semantics of import settings

Full Import		aux F { ref = A }
Init Import		level F { init = A }
Tail Import, Init Export		level A { init = F } level F { init = <...> ref = A }

From the above, it can be seen that in the cases of full import and tail import, the formal and the actual parameter will actually share the memory of the actual parameter. For full export and tail export, the parameters will share the memory of the formal parameter. This means that it is only initial import and initial export that actually will copy values between the two sides, and the copying takes place only during initialization of a simulation run.

Passing parameters by reference instead of by value is necessary in order to allow for import and export of levels. As an additional bonus, memory requirements are reduced (shared memory) and execution speed increased (no copying of values).

- **A conveyor belt component**

Let us use the mechanisms described above to convert our previous conveyor model example into a component that is used as a submodel of another component. In the figure below, we have created a component called *Conveyor*, based on the *Transport* submodel in Figure 17.

```

mainmodel Conveyor {
  unit Rate {
    def = external
  }
  aux Input {
    public
    import = full
    inflow
    unit = .Rate
  }
  aux Duration {
    public
    import = full
    unit = __time
  }
  aux Output {

```

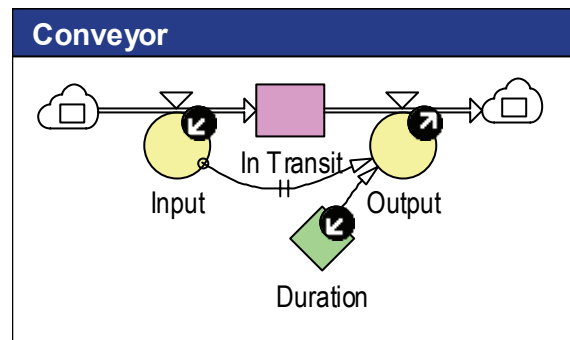


Figure 29: Conveyor component

Input and *Duration* are imported to the component from the outside, while *Output* is

```

public
export = full
outflow
def = DELAYMTR(Input, Duration)
unit = .Rate
}
level 'In Transit' {
  init = Input * Duration
  flows = +dt*Input - dt*Output
  unit = Rate*__time
}
}

```

computed by the component, and exported.

The level *In Transit* is strictly not necessary for the implementation, but helps readability.

A general component like this should be possible to use with any measurement unit for the actual parameters *Input* and *Output*. This is achieved through the local unit *Rate*, which is defined to be **external**, i.e., determined outside the component.

The *Conveyor* component can be used by another component by following these steps.

1. Create a new component (Main).
2. Drag the *Conveyor* component from the component catalog and drop its symbol into *Main's* diagram. A submodel named *Conveyor 1* will be created inside *Main*, with interface settings copied from *Conveyor*.
3. Rename *Conveyor 1* to *Transport*.
4. Create the levels *Remote Inventory* and *Local Inventory*, and connect up the flows as illustrated in the figure below.

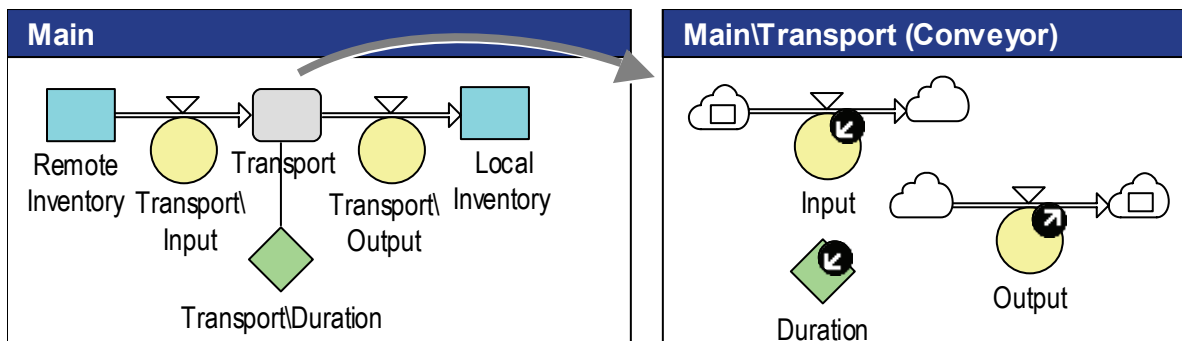


Figure 30: Hierarchical transport model using conveyor component

Note that the diagram of *Transport* contains only the interface variables, and nothing from the implementation of *Conveyor*. The fact that *Transport* is implemented using *Conveyor* is reflected in the title bar of *Transport's* diagram. The actual connection between *Transport* and *Conveyor* is a property of the *Transport* variable, and can be set either via a property page or the equations view of the model.

- **Components can deduce measurement units from actual parameters**

The measurement units of component variables can be selected either from the set of global units or from local units defined inside the component. If we make a general component, like the conveyor, it is impossible to know the measurement units of the actual parameters at the time of component construction. We could, of course, avoid specifying units altogether, but this is not a good solution, as explicit units can significantly improve the quality of a model as well as input and output of values. For this reason, we have introduced the special unit expression **external**, which can be used when defining local units inside a component. An external unit is treated like

an **atomic** unit when the component is run as a stand-alone model. When a new instance of the component is created inside another model, the definitions of the external units are deduced from the units of the actual parameters to the component.

Let us assign the unit *Product* to the level variables *Remote Inventory* and *Local Inventory* in the previous example. This will imply that the variables *Transport\Input* and *Transport\Output* get the unit *Product/___time*. (The reserved unit name **___time** denotes the time unit of the simulation.) Inside the *Transport* instance of *Conveyor*, the external unit *Rate* will be mapped to *Product/___time*. The unit of *In Transit* will be mapped to *Product*, using the following equation:

$$\text{unit 'In Transit'} = \text{Rate} * \text{___time} = (\text{Product} / \text{___time}) * \text{___time} = \text{Product}$$

A period (.) in front of a unit name selects the local unit scope. See Appendix A for more details about measurement units.

- **Default definitions and optional parameters**

When the actual and the formal parameters of a component are paired together, there may be situations where some parameters end up without a counterpart. This could be treated as an error, but instead we will use this situation to allow for default definitions and optional parameters.

If an actual parameter is not connected to a formal parameter, the definition of the actual parameter will be used as a default definition for that parameter. Correspondingly, a formal parameter that is not matched with a corresponding actual parameter will use its own definition.

One consequence of this is that a component can export optional values. In the conveyor example, for example, the value of the *In Transit* level can be made available. It is up to the submodel that instantiates the conveyor to use the exported value or not.

When a component is run as a stand-alone model, it has no actual parameters. In this case, definitions for imported variables can serve as test definitions.

If an imported formal parameter does not have a definition, it must be connected to a corresponding actual parameter in order to complete the connection. Correspondingly, if an exported actual parameter does not have a definition, it must be connected to a formal parameter.

6.3 Component Model Example

In this paragraph we will create a component version of the *Market* and the *Retailer* of the example in chapter 5.2. We start out with the market, and create a component named *Market* inside the component catalog of our simulation project file. The diagram for the *Market* submodel in Figure 24 can be copied directly into the *Market* component, and edited like this:

1. Set the component interface property to *Market*.
2. Make *Supplies* full import and *Orders for Supplies* full export.
3. In order to make the component more flexible, we expose more of the parameters for optional initialization and inspection from the outside. *Potential Customers*, *Customers*, *Products per Customer* and *Encounter Efficiency* should be possible to initialized from the outside.

The resulting diagram looks like this:

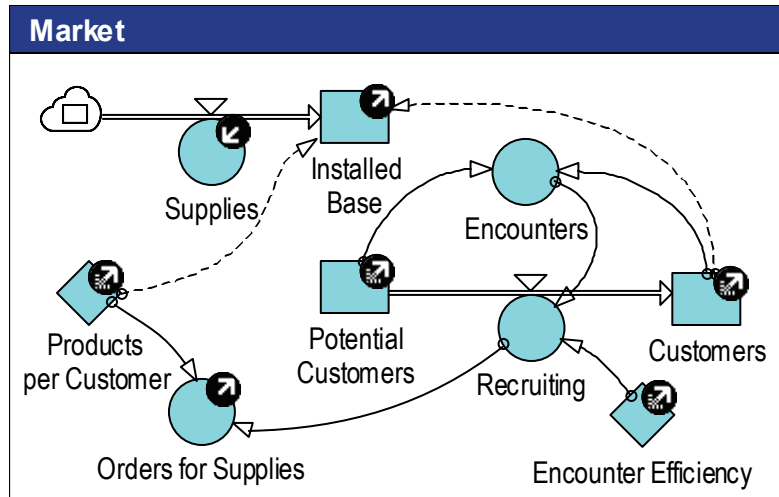


Figure 31: Market component

The equations for the above component are given below. Modified lines are boxed.

```

mainmodel Market {
  level Customers {
    public; import=init; export=tail; readonly
    init = 1 <<Person>> // Default definition
    flows = -dt*Recruiting
    doc = "Number of actual customers."
    unit = Person
  }
  aux 'Encounter Efficiency' {
    public; import = init; export = tail
    init = 0.1 <<Person/Encounter>> // Default definition
    doc = "Recruited customers per encounter"
    unit = Person/Encounter
  }
  aux Encounters {
    unit = Person/Day
    def = Customers * 'Potential Customers'/(Customers + 'Potential Customers')
    doc = "Number of encounters between customer and non-customer in a day."
    unit = Person/Day
  }
  level 'Installed Base' {
    public; export = full; readonly
    init = Customers * 'Products per customer'
    flows = +dt*Supplies
    doc = "Number of products in the market."
    unit = Product
  }
  aux 'Orders for Supplies' {
    public; export = full
    def = Recruiting * 'Products per Customer'
    doc = "Product orders"
    unit = Product/Day
  }
}

```

```

level 'Potential Customers' {
  public; import = init; export = tail; readonly
  init = 1000 <<Person>> // Default definition
  flows = -dt*Recruiting
  doc "Number of potential customers."
  unit = Person
}
aux 'Products per Customer' {
  public; import = init; export = tail
  init = 1<<Product/Person>> //7 Default definition
  doc = "Average number of products per customer."
  unit Product/ Person
}
aux Recruiting {
  def = Encounters*'Encounter Efficiency'
  doc = "Customer orders."
  unit = Person/Day
}
aux Supplies {
  public; import = full; inflow
  def = DELAYPPL(Orders, 1<<Day>>) // Test equation
  doc = "Products received."
  unit = Product/Day
}
}

```

An even more general version of the component would define *Product* as an **external** unit of the *Market*.

In a similar way as for the *Market*, we can create a component version of the *Retailer*. In preparing the component for re-use, we call it *Provider* instead of *Retailer*. This way the component can be used also to create a *Wholesaler*, for example. We copy the diagram for the *Retailer* submodel in Figure 22 into a new component called *Provider*, and make the following changes to the component:

1. Set the component interface property to *Provider*.
2. Make *Orders* and *Supplies* full import and *Shipments* full export.

The diagram for *Provider* now looks like this:

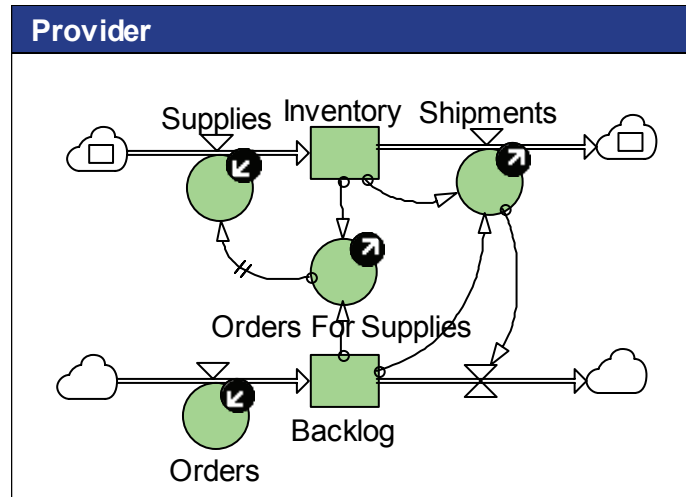


Figure 32: Provider component

The corresponding equations are given next:

```

mainmodel Retailer {
  level Backlog {
    init = 0 <<Product>>
    flows = -dt*Shipments +dt*Orders
    doc = "Orders not yet fulfilled."
    unit = Product
  }
  level Inventory {
    init = 0 <<Product>>
    flows = -dt*Shipments +dt*Supplies
    doc = "Products ready to ship to customers."
    unit = Product
  }
  aux Orders {
    public; import=full
    doc = "Product orders"
    unit = Product/Day
  }
  aux 'Orders For Supplies' {
    public; export=full
    def = MAX(0 <<Product/Day>>, Backlog - Inventory)
    doc = "Orders for more supplies."
    unit Product/Day
  }
  aux Shipments {
    public, export=full, outflow
    def = MIN(Inventory, Backlog)
    doc = "Product shipments to customers."
    unit = Product/Day
  }
}

```

```

aux Supplies {
  public; import=full; inflow
  def = DELAYMTR('Orders For Supples', STARTTIME + 4 <<Day>>)
  doc = "Received products from supplier."
  unit = Product/Day
}
}

```

The two models can now be connected together, forming our *Fourth Model*, which looks exactly like the *Third Model* in Figure 23.

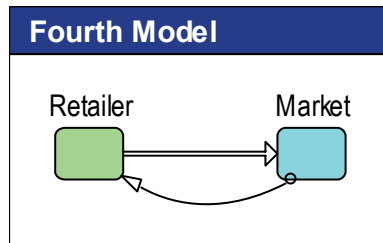


Figure 33: Fourth Model

The equations for the *Fourth Model* are significantly simpler than for *Third Model*, which does not use components. In the equations listing below we have omitted documentation and interface variables that are not used.

```

mainmodel 'Fourth Model' {
  submodel Retailer {
    interface = Provider
    implementation = Provider // Component that implements the retailer
    aux Orders {
      public; import=full
      ref = ..Market\Orders for Supplies'
      unit = Product/Day
    }
    aux Shipments {
      public; outflow; export=full
      unit = Product/Day
    }
  }
  submodel Market {
    interface = Market
    implementation = Market // Component that implements the market
    aux 'Orders for Supplies' {
      public; export=full
      unit = Product/Day
    }
    aux Supplies {
      public; inflow; import=full
      ref = ..Retailer\Shipments
      unit = Product/Day
    }
  }
}
}

```


The finished model has only six variables; two submodels with two parameters each for exchanging information. Below the surface, there is a much richer structure, as depicted by the full variable hierarchy below:

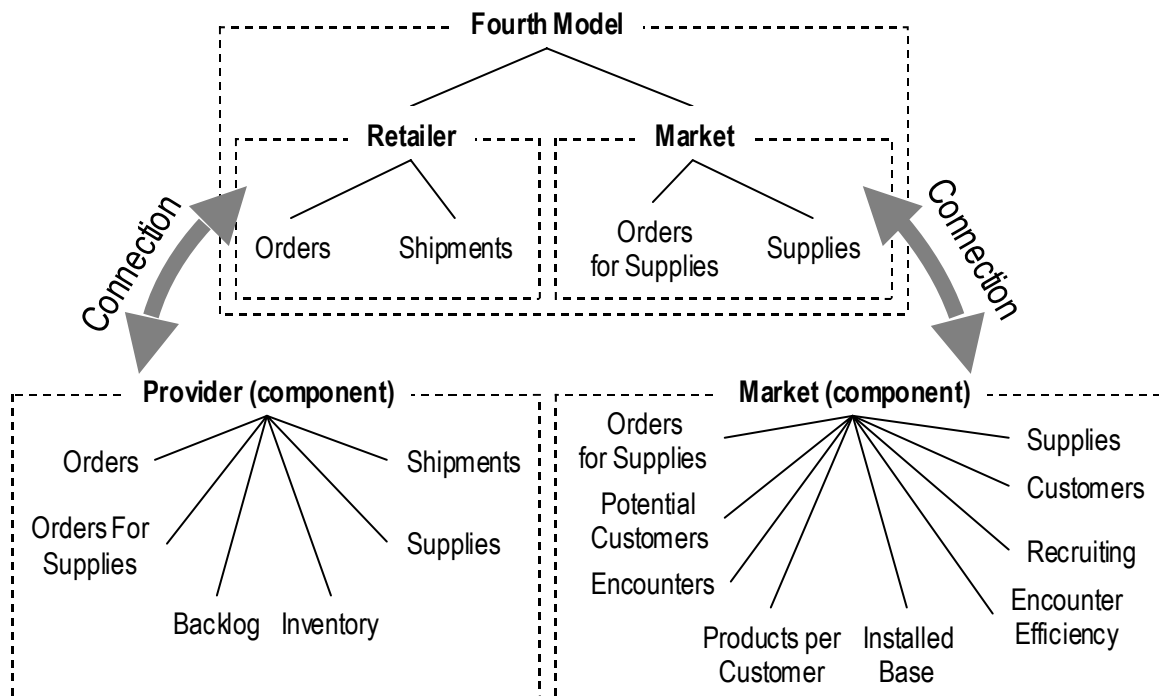


Figure 34: Variable Hierarchy of Fourth Model

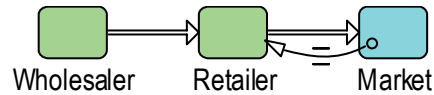
6.4 Multiple Instances of a Component

In the previous chapter we created two re-usable components, the *Market* and the *Provider*. We can use these two components to create a model of a wholesaler that delivers products to a *Retailer*, which in turn sells to a market. The steps in creating such a model are given below:

- | Step | Resulting diagram |
|---|-------------------|
| 0. Create a new component and call it <i>Fifth Model</i> . | |
| 1. Drag an instance of the <i>Provider</i> component into the diagram and call it <i>Wholesaler</i> . | |
| 2. Drag another instance of <i>Provider</i> into the diagram and call it <i>Retailer</i> . | |
| 3. Drag an instance of the <i>Market</i> component into the diagram. | |
| 4. Select the flow tool and drag a flow from Wholesaler to <i>Retailer</i> . The system will ask you to confirm that you want to create a flow from <i>Wholesaler\Shipments</i> to <i>Retailer\Supplies</i> . | |
| 5. Select the flow tool again, and drag a flow from <i>Retailer</i> to <i>Market</i> . The system will ask you to confirm the creation of a flow from | |

Retailer\Shipments to Market\Supplies.

6. Select the link tool, and drag a link from Market to Retailer. In the pop-up that appears, pick the entry that links *Market\Orders for Supplies* to *Retailer\Orders*.



7. Again, select the link tool, and drag a link from *Retailer* to *Wholesaler*. Pick the link from *Retailer\Orders for Supplies* to *Wholesaler\Orders*. The final result is displayed below.

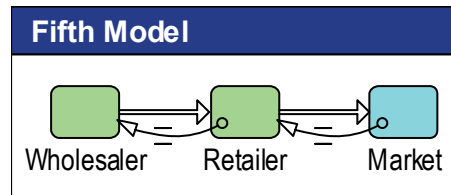


Figure 35: Fifth Model

In these seven steps we have created a model with three interconnected sub-models. Two of the sub-models are instances of the same component, the *Provider*. The third instance is of the *Market* component. Altogether, we have created a system with 21 basic variables (2 x 6 from the *Provider* and 9 from the market), all fully defined, documented and equipped with the correct measurement units. The variable hierarchy for this example is like this (details inside the *Provider* and *Market* components are omitted):

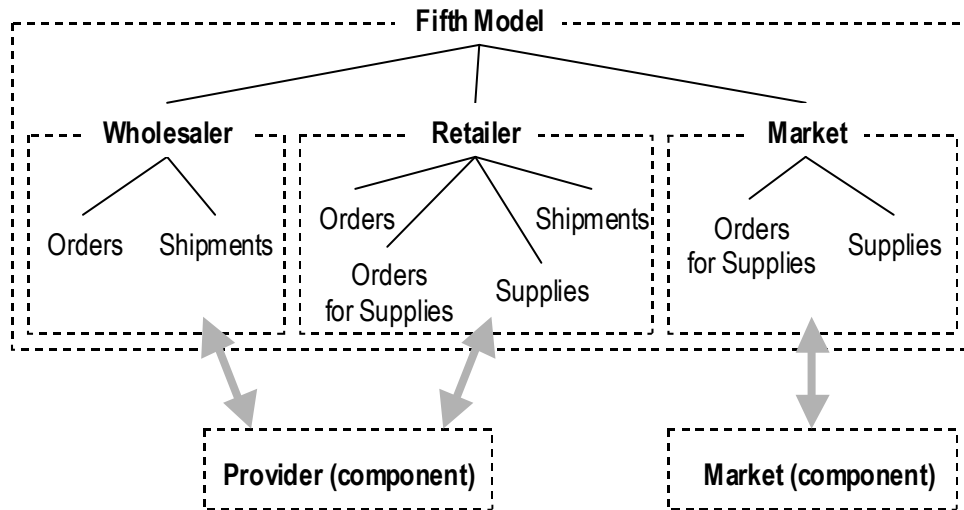


Figure 36: Variable hierarchy of Fifth Model

The inventory management policy of the *Provider* component is quite simple. If we make improvements to *Provider*, they will immediately take effect on the *Wholesaler* and the *Retailer* of the above example.

But, instead of doing that, let us see how the inner workings of the *Retailer* can be changed without changing the behavior of the *Wholesaler*, or any other part of the model. The solution builds on the concept of polymorphism, and is described in the following chapter.

6.5 Polymorphism and component swapping

As long as components communicate via interfaces and do not make use of the implementation of one another, a component can be exchanged for another component with the same interface. This is called component swapping. Let us create a more advanced *Provider* component, with an inventory management policy that takes into account the number of products that are on order from the supplier, and the expected demand. The easiest way to do this is to make a copy of the *Provider* component, and make the modifications that are described below.

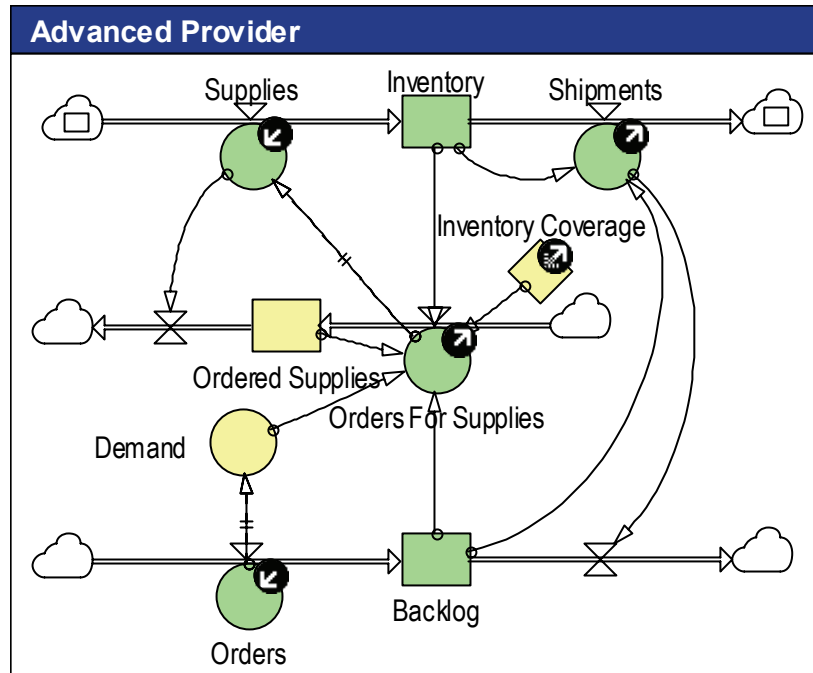


Figure 37: Provider component with improved ordering policy

The equations that are added or modified from the original Provider version are given below:

```

level 'Ordered Supplies' {
  init = 0 <<Product>>
  flows = +dt * 'Orders for Supplies' - dt * Supplies
  unit = Product
}
aux 'Inventory Coverage' {
  public; import=init; export =tail
  init = 3 <<Day>>
  unit Day
}
aux 'Demand' {
  def = DELAYINF(Orders, 10 <<Day>>)
  unit Product/Day
}
aux 'Orders for Supplies' {
  public; export=full
  def = MAX(0 <<Product/Day>>,
    'Inventory Coverage' * Demand - (Inventory + 'Ordered Supplies' - Backlog))
  unit Product/Day
}

```

It takes only one small change to *Fifth Model* of the previous chapter in order to create the *Sixth Model*, where the *Wholesaler* is a normal *Provider* and the *Retailer* an *Advanced Provider*.

1. Open up the property page of *Retailer*, and change the implementation property from *Provider* to *Advanced Provider*.

Since *Provider* and *Advanced Provider* have the same interfaces, the above is the only step that needs to be taken. In response to this command, the system will do the following steps automatically:

2. Disconnect *Retailer* from its instance of *Provider* and destroy that instance.
3. Connect *Retailer* to a newly created instance of *Advanced Provider*.

The resulting variable hierarchy is given below:

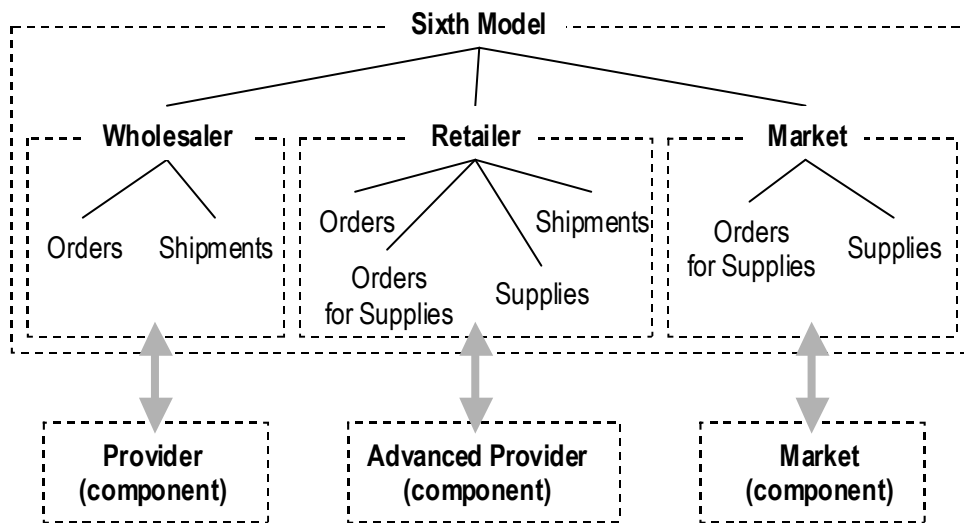


Figure 38: Variable hierarchy of Sixth Model

6.6 Components as User-defined Functions

A component can be used as a function by marking one of its parameters with the return attribute. Here is an example where SUMOF is defined as a component that computes the sum of its arguments.

```

mainmodel SUMOF {
  aux A { import=full }
  aux B { import=full }
  aux C { export=full; return; def=A + B }
}

```

The SUMOF model can be used as a submodel, but it can also be used as a user-defined function, for example like this:

```

aux X { def=SUMOF(P, Q) }

```

When using a component as a function, the compiler will generate an instance inside the current basic variable. The instance name will be the same as the function name, which is the same as

the component name. If the same component is used twice in the same definition, sequence numbers will be added to the implicitly defined components.

The above definition of X is equivalent to:

```

aux X {
  submodel SUMOF {
    private
    aux A { import=full; ref=..P; }
    aux B { import=full; ref=..Q; }
    aux C { export=full; return }
    implementation SUMOF;
  }
  def = .SUMOF\C; // Set value equal to return parameter of component
}

```

The above solution has the advantage over macros that code will be re-used for all instances of the function. The solution also allows for browsing into the implementation of the function in the same way as for normal submodels.

6.7 Foreign Components

The encapsulation mechanisms implied by the separation between interface and implementation of a component, opens up the possibility to create special types of components that are implemented as non-SD models.

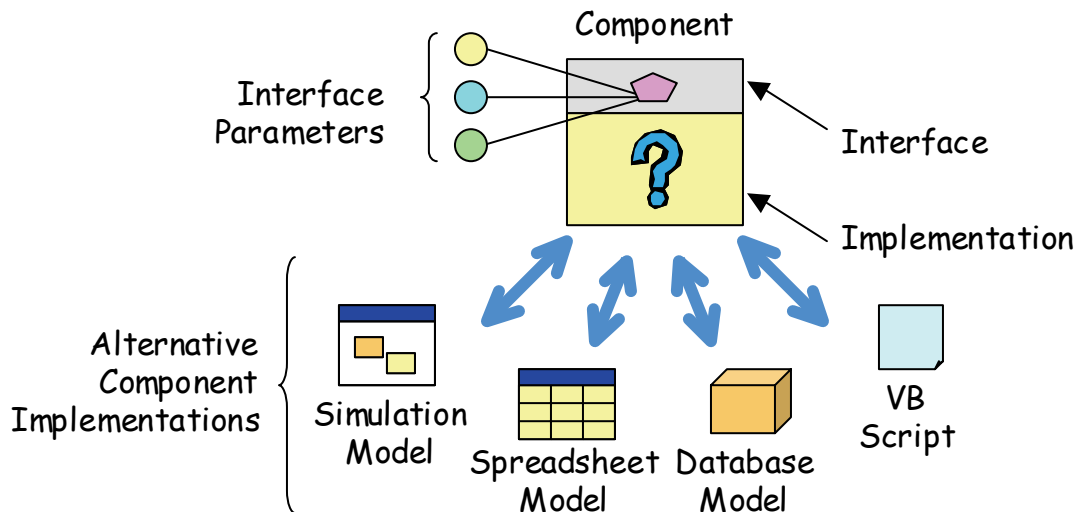


Figure 39: The component concept can be expanded to include other modeling technologies

In the above scenario we have included spreadsheets, databases and Visual Basic models in addition to the simulation models as ways of implementing components. Conceptually, any software object that can import and export values for variables can be included in this framework. Potential uses of this approach are listed below:

- Existing spreadsheet models can be embedded into simulation models
- The macro language of the spreadsheet software can be used to create custom functions to be used by the simulation

- Analysis and presentation features of a spreadsheet can be accessed from a simulation
- Simulations can be initialized from databases, scenarios can be read and simulated results written to the database.
- Visual Basic can be used to create user-defined functions.
- Visual Basic can be used to access any other resource that is available from the computer that hosts a running simulation.

7 Connections

Purpose of solution: Coping with complexity of object relationships through bundling of parameters. Type-safe and guided connection process for components.

Core of solution: Objects can have sockets and plugs that define possible connection points for model objects.

Object-oriented concepts: interface, object relationship

Any system can be described from a set of objects and object relationships. In the previous chapters we have dealt with object-oriented definitions for defining objects, interfaces (types), and implementations (classes). We have seen how typed interfaces can be used to introduce a type-safe swapping of components in order to customize a model or test out the consequences of alternative solutions.

This chapter deals with the object relationship part. The same way typed interfaces serve as bridges between objects having interfaces and classes implementing interfaces, it should be possible to define type-safe connections between objects. The idea is that if one component supports one end of a given connection type, and another component supports the other, the two can be connected.

- **Weaknesses of low-level variable connections**

This far we have worked with parameters that are basic variables, i.e., levels, auxiliaries or constants. Such parameters are connected up using basic link and flow symbols, or the special reference link that we have introduced. When two basic variables are connected (either explicitly or implicitly via a connection to a component), it is possible to make low-level tests to verify a connection. Such tests include the following:

- Data type check (the data type of the actual and formal parameters must match)
- Unit check (the unit of measure of the actual and formal parameters must match)
- Dimension check (the array dimensions of the actual and formal parameters must match)
- Import and export settings check (the actual and formal parameters must have the same import and export settings)
- Variable type check (the variable type of the actual and formal parameters must be valid for the given import and export settings)

The above checks will identify invalid parameter combinations at a technical level. But, even if a connection is mathematically possible to do, the connection may not make sense in the context

where it is used. As an example, there is nothing that prevents us from rerouting the orders for supplies in Figure 35 to the following, erroneous version:

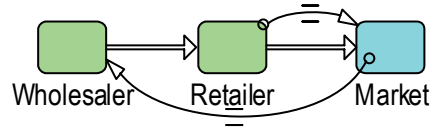


Figure 40: Example of erroneous Connections that will be silently accepted by the system

When connecting sub-models, there will normally be more than one basic parameter involved. In the above example, there are two parameters involved in the connection between *Market* and *Retailer*, and another pair of parameters is involved between *Retailer* and *Wholesaler*.

- **A socket-wire-plug solution to variable connections**

What we need is a way to bundle parameters together into a connector that can be plugged into a compatible connector of another component. Two new structured variable types are introduced for this purpose, the sockets and the plugs. Sockets and plugs can hold parameters, and they also have an interface type. A plug can be connected to socket if the plug has the reverse interface of the plug. When a plug and a socket are connected, the parameters on each side are connected up in the same way as when a submodel variable is connected to a component.

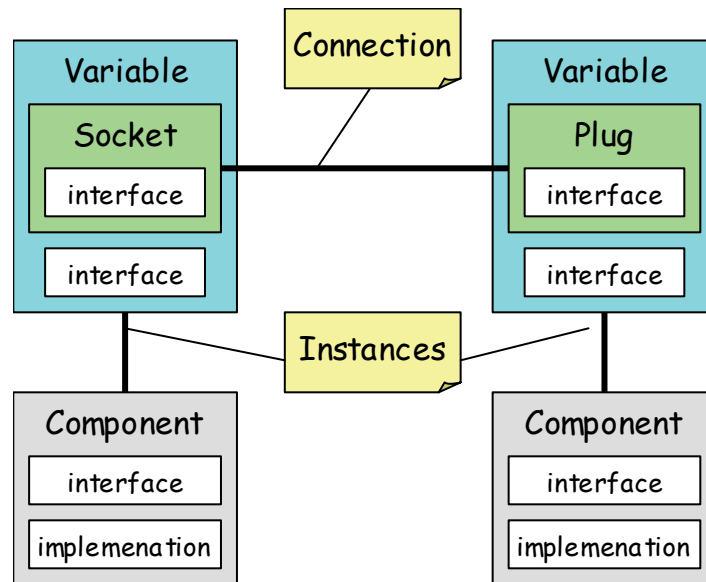


Figure 41: Connections can use interfaces, the way variables relate to components via interfaces

In both cases the connection can only be made if the interfaces match. Also, when a connection takes place, the implications for the underlying model equations are determined automatically. (It is not necessary to edit any equations for the connection to take place).

The socket symbol looks like this  and the plug symbol like this .

Let us start by making a wired version of the *Provider* component. To do this, we create a copy of *Provider*, and call it *Wired Provider*. Then we create a socket *Supplier* for connecting to the *Wholesaler*. We call the connector interface *Supply Connector*, and it includes two parameters, one for sending out orders, and one for receiving products. The equations for the *Supplier* socket looks like this:

```

socket Supplier {
  interface = 'Supply Connector'
  aux Orders {
    public; export = full
    doc = "Orders for more supplies."
    unit = Product/Day
  }
  aux Shipments {
    public; import = full
    doc = "Received products from supplier."
    unit = Product/Day
  }
}

```

Copy the *Supplier* socket and transform it into a plug and call it *Market*. When a socket is transformed into a plug, the interface is automatically reversed by reversing **import** and **export** settings (and exchanging **inflows** and **outflows**). The *Market* plug becomes like this (after editing of the **doc** properties):

```

plug Market {
  interface = 'Supply Connector'
  aux Orders {
    public; import = full
    doc = "Orders from customer."
    unit = Product/Day
  }
  aux Shipments {
    public; export = full
    doc = "Shipments to customer."
    unit = Product/Day
  }
}

```

The *Wired Provider* component looks like this after the modifications are complete:

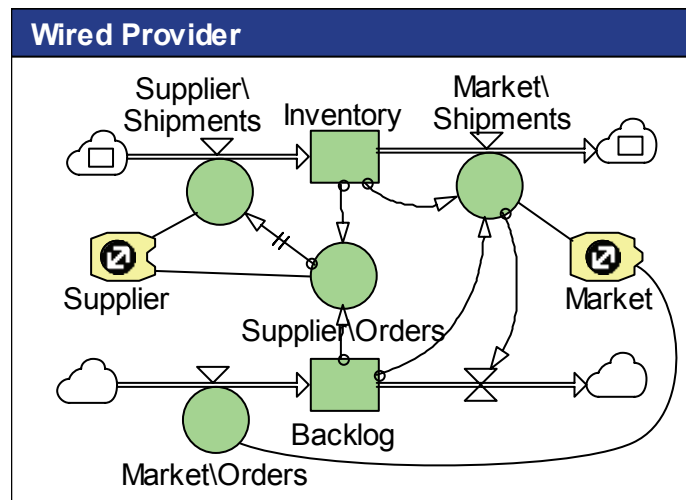


Figure 42: Provider component with socket and plug connectors

The component now has only two parameters, the *Supplier* socket and the *Market* plug. This makes the interface simpler, and connections easier. Let us do similar changes to the *Market*

component. The *Supply Connector* interface can be used again, and we simply put a *Retailer* socket into the *Wired Market* component:

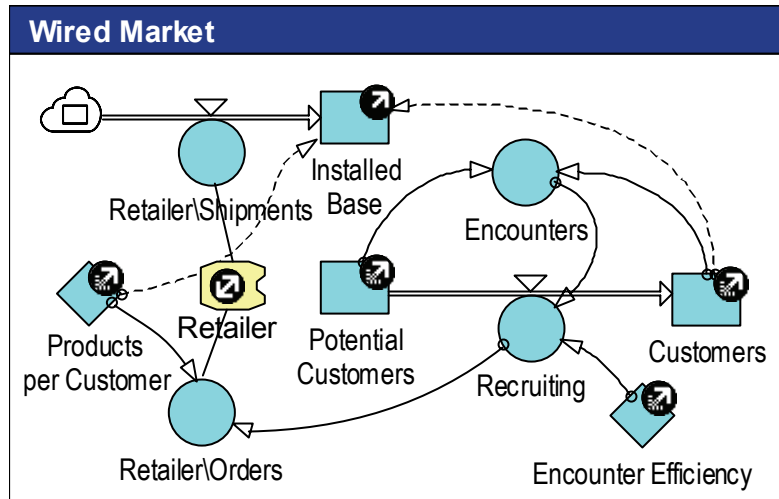


Figure 43: Market component with socket connector

With the above component in place, we can create our final example, the *Seventh Model*. The model will be a wired version of the model in Figure 35. The steps in creating the full model from the two components are given below.

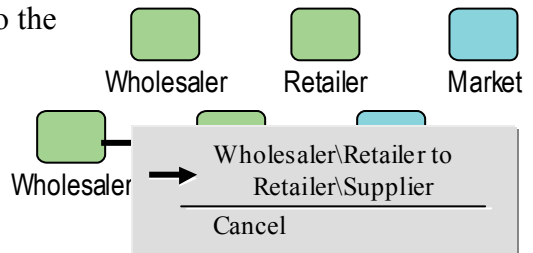
Step	Resulting diagram
------	-------------------

0. Create a new component and call it *Seventh Model*.

1. Drag an instance of the *Wired Provider* component into the diagram and call it *Wholesaler*.

2. Drag another instance of the *Wired Provider* into the diagram and call it *Retailer*.

3. Drag an instance of the *Wired Market* component into the diagram and call it *Market*.



4. Select the link tool and drag a link from *Wholesaler* to *Retailer*. The system will ask you to confirm that you want to connect *Wholesaler\Retailer* to *Retailer\Supplier*.

5. Select the link tool again, and drag a wire from *Retailer* to *Market*. The system will ask you to confirm the creation of a wire link from *Retailer\Market* to *Market\Retailer*. The model is now finished, and looks like the diagram below.

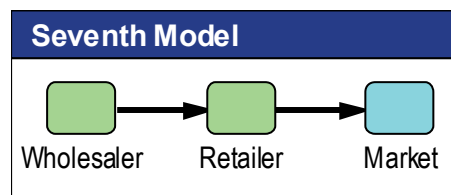


Figure 44: Seventh Model

A link from a plug to a socket is called a wire link, and it is displayed using a solid line. Since sockets and plugs can hold more than one parameter, wire links normally carry several values at

the time, some in one direction, and other is the opposite direction. The direction of the wire link is somewhat arbitrarily chosen to point from the plug to the socket (the plug is inserted into the socket).

A plug is connected to a socket using the **def** statement of the plug, as illustrated by the equations for the *Seventh Model*, below:

```
mainmodel 'Seventh Model' {
  submodel Market {
    interface = 'Wired Market'
    implementation = 'Wired Market'
    socket Retailer {
      public, import, export
      interface = 'Supply Connector'
      def = ..Retailer\Market
      ... parameters of socket omitted
    }
  }
  submodel Retailer {
    interface = 'Wired Provider'
    implementation = 'Wired Provider'
    socket Supplier {
      public, import, export
      interface = 'Supply Connector'
      def = ..Wholesaler\Retailer
      ... parameters of connector omitted
    }
    plug Market {
      public, import, export
      interface = 'Supply Connector'
      ... parameters of connector omitted
    }
  }
  submodel Wholesaler {
    interface = 'Wired Provider'
    implementation = 'Wired Provider'
    plug Retailer {
      public, import, export
      interface = 'Supply Connector'
      ... parameters of connector omitted
    }
  }
}
```

The above model consists of only three variables and two (wire) links at the topmost level. These three variables are (sub) models, and they can only be connected in ways that obey the types and polarities of the connector parameters (sockets and plugs). The wiring process is done without every entering any equations in order to connect the components together. This significantly reduces the complexity of model building and the risk of making wrong connections. It is a requirement however, that an appropriate set of components is available. Around the need for ready-made building blocks we may experience a growing market, and a further expansion of the field of SD into communities that find traditional SD modeling too difficult.

7.1 Wire Flows

Links between a socket and a plug are called wire links, and they are displayed using a thick line. It turns out that there are places where wire flows come in naturally. As an example, let us look at a resource that has multiple consumers. At the resource end, we can make a connector with one import parameter for receiving requests and one output parameter for delivering the requested amount of the resource. Similarly, the consumers will have a connector with one export parameter for making requests and one import parameter for receiving resources that are granted. A diagram for this model looks like this:

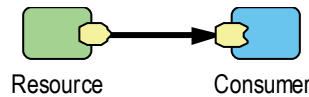


Figure 45: Resource and Consumer Model

Now, if there are multiple consumers, and the number of requests gets larger than the available resource, we need to prioritize the requests. The prioritizing may deny some requests, or grant a smaller amount than requested if the sum of the requests exceed the capacity of the resource.

Prioritization is not a natural part of a resource, and not a part of the consumer either. Instead, we want a resource manager component to take care of resolving conflicting requests. The resource management may have opinions about the relative importance of consumers, the availability of other resources that are also needed, etc.

What we need is a prioritization component that can receive multiple requests for a resource, adjust the requests as needed in order to stay within the capacity of the resource, and channel granted resources to consumers. As a first version, let us create a resource manager component with two connectors, one for connecting to the consumer and one for connecting to the resource. The model can now be expanded like this:

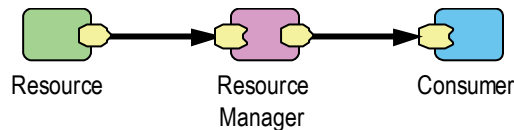


Figure 46: Resource and Consumer Model with Resource Manager

Relate the above to the definition of a regular flow between two levels, as displayed below.

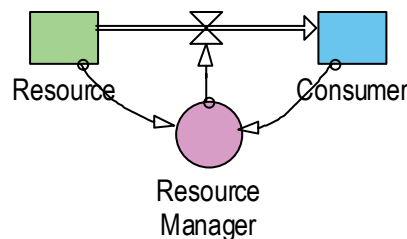


Figure 47: Resource and Consumer model using basic variables, links and a flow

The variable controlling the flow has been pulled away from the flow valve in order to stress the fact that the flow actually represents two links departing from the rate in the middle. One link goes to the level at the tail of the flow, and the other one goes to the level at the head.

The diagram below is equivalent to the flow version, given that *Resource* and *Consumer* are defined using the INTEGRATE function, which essentially puts a level inside an auxiliary:

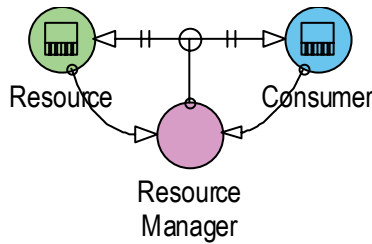


Figure 48: Resource and Consumer model using basic variables and only links

The above observations lead naturally to the definition of a wire flow as a connection of four connectors: one at each end, and two at the valve. We can now create a flow version of the model in Figure 46.

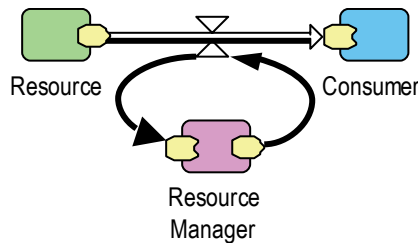


Figure 49: Resource and Consumer connected by wire flow

As always, the connectors can be hidden from view, and the controlling variable snapped on to the flow valve, like this:

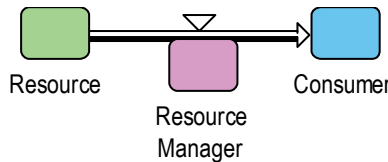


Figure 50: Hidden connector view of Resource and Consumer connected by wire flow

Wire flows can be used only if the socket at the tail matches the plug at the valve, and the socket at the valve matches the socket at the head.

7.2 Arrayed Components and Connectors

The use of arrays is not the main focus of this paper. But, in the context of the above example, the question comes to mind: How do we model a system where multiple consumers make use of a shared resource?

- **Arrayed Components**

A simple solution is to make the *Consumer* into an array. This can be done by opening up *Consumer's* property page and editing the dimensions field. The equations will be adjusted automatically, like this:

```

submodel Consumer {
  dim = 1..3           // Three customers
  ... // rest omitted
}

```

The component implementing *Consumer* does not need to be an arrayed component. When a submodel variable specifies dimensions, all variables of this instance of the associated component will get extra (initial) dimensions.

- **Arrayed Connectors**

Another solution is to make the plug at the *Resource Manager* into a dynamic array. A dynamic array is much like an external unit, in that it gets specified from the context. This feature makes it possible to create a diagram like the one below:

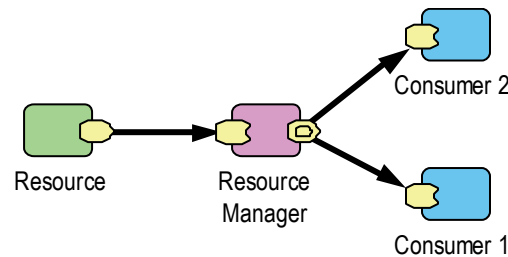


Figure 51: Managing a resource with multiple consumers

The following list compares the single-consumer model in Figure 46 (and in Figure 49) to the above multi-consumer version:

Variable or Component	Comment
Resource Component and Variable	Unchanged
Consumer Component and Variable	Unchanged
Resource Manager Interface	Plug changed from scalar to dynamic array.
Resource Manager Component	Implementation changed to deal with multiple consumers.

7.3 Co-flows

Objects have attributes. As an example, a worker can have a given level of experience. The OO term for an attribute is an object property. The value of a property is part of an object’s state.

Object Oriented Software Model	System Dynamics Model
<div style="border: 1px solid black; background-color: #ffffcc; padding: 2px; margin-bottom: 2px;">Jim:Person</div> <div style="border: 1px solid black; background-color: #cccccc; padding: 2px; margin-bottom: 2px;">Experience=80</div> <div style="border: 1px solid black; background-color: #ffffcc; padding: 2px; margin-bottom: 2px; margin-left: 20px;">Bob:Person</div> <div style="border: 1px solid black; background-color: #cccccc; padding: 2px; margin-bottom: 2px; margin-left: 20px;">Experience=84</div> <div style="border: 1px solid black; background-color: #ffffcc; padding: 2px; margin-bottom: 2px; margin-left: 40px;">Jane:Person</div> <div style="border: 1px solid black; background-color: #cccccc; padding: 2px; margin-bottom: 2px; margin-left: 40px;">Experience=88</div>	<div style="border: 1px solid black; background-color: #ffffcc; width: 40px; height: 30px; display: flex; align-items: center; justify-content: center; margin-bottom: 10px;">3</div> <div style="margin-bottom: 10px;">Workers</div> <div style="border: 1px solid black; background-color: #cccccc; width: 40px; height: 30px; display: flex; align-items: center; justify-content: center; margin-bottom: 10px;">84</div> <div style="margin-bottom: 10px;">Average</div> <div style="margin-bottom: 10px;">Experience</div>

Figure 52: Attributes of individual objects versus average attributes of object groups

In SD models, we normally do not represent objects one-by-one. Instead, object counts are stored in level variables. When there is a need to keep track of object properties, this is done by having

separate levels. Typically, an attribute level stores the average attribute value (or the sum of the attribute values) of the objects in a corresponding level variable.

In order to work with attributes of groups of objects (like the experience level of a workforce), we need to add extra levels to hold the attributes. In addition we need to update the attribute levels to match the changes that take place in the group. The updating of group attributes can be done through a technique that is called co-flows. A co-flow is a parallel flow that carries attributes for the items that flow in the main flow.

Working with co-flows in basic SD diagrams is not straightforward. The number of extra variables tends to grow rapidly, increasing the complexity of the diagram. In addition, the logic for co-flows can be quite complicated.

Diagrams and equations for modeling groups of objects and their (average) attributes can be simplified through the use of submodels, sockets, plugs, and wire flows. A single submodel variable can hold the object count and any number of attribute variables. Sockets and plugs can define connection points for incoming and outgoing object flows and their associated (average) attributes. Wire flows can transfer objects and attributes together.

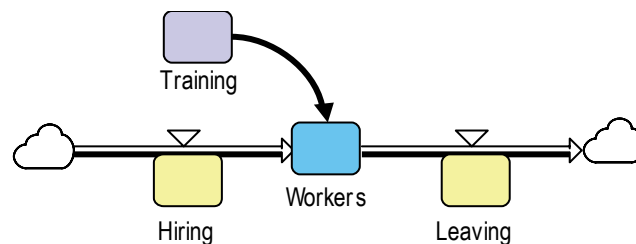


Figure 53: Flows and levels workforce with attributes

The above diagram indicates how to create a model of a workforce with one or more attributes, such as experience. People are hired with a given (average) set of attributes, and enter the workforce. While in the workforce, people get experience through training and other activities (practice). When leaving, workers bring along their experience, influencing the attributes and the count of remaining workers.

8 Additional Model Views

A hierarchical view of a model is good for presenting the structure at different levels of abstraction. The division of models into submodels matches naturally the way we organize systems into subsystems. The hierarchical view is not good for showing feedback relationships, however. Feedback typically follows paths of influences that cross the borders of sub systems. When we look at a model of a subsystem alone, the sections of feedback loops that extend beyond the boundary of the submodel gets cut away. When a loop is cut, it is not a loop any longer, and the feedback path vanishes out of view.

The model-view metaphor displayed in Figure 4 calls for diagrams of more than one type. The variable relationships of a model in the extended SD language can be mapped to relationships between individual, basic variables. The semantics of the structured variables is expressed in terms of basic variables, so a model can still be looked upon as a model with levels and rates only.

This means that we can introduce a causal-loop view of any model that is made in the extended SD language. In this view, a diagram is not constrained to displaying variables that are siblings. In the causal-loop diagram view, variable dependencies can be traced up, down and across inside the variable hierarchy, forming the paths that constitute the feedback loops that, together with system state, generate the behavior mode of the model at any given point in time. Rule three in Table 1 needs to be specified differently for causal-loop diagrams than for accumulator-flow diagrams (see Table 2). This task is left as an exercise for the interested reader. In Myrtveit and Saleh (2000) it is described how the dominant behavior mode and the contributing structural links can be superimposed on a causal-loop diagram. The figure below illustrates this for a simple population model.

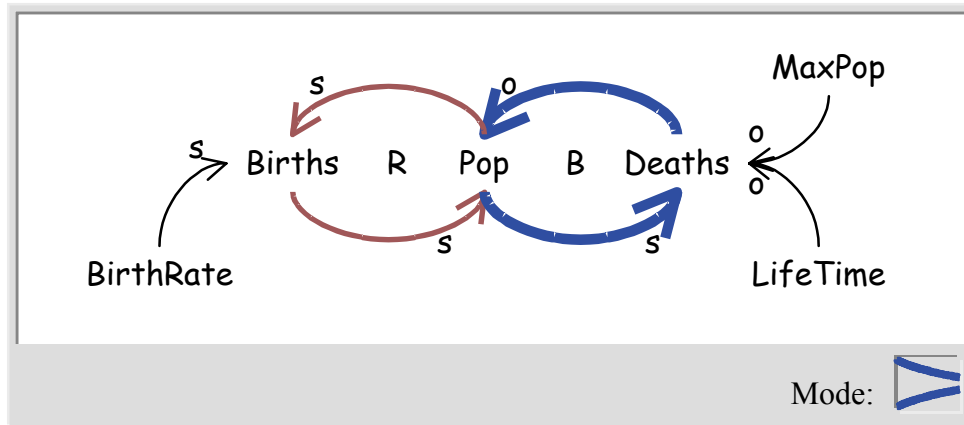


Figure 54: Diagram showing relative contribution of substructures to the dominant behavior of a model

The above diagram shows the simulation in a state where the population is approaching the maximum capacity of the environment. The balancing loop (B) is contributing the most to the current mode of behavior, which is a convergent mode, illustrated by the icon in the lower right-hand corner of the figure.

9 Acknowledgments

The ideas in Appendix A about measurement units have been jointly developed by Arne Kråkenes and myself, mainly during a two-day stay at the old mining town of Røros in 1998. The object-oriented solutions described in the rest of this paper were conceived during three summer weeks together with my family at Dewey Beach (Pennsylvania). Arne and the rest of the Powersim team have given valuable feedback on the initial drafts. I am very impressed and pleased by the time, effort and dedication that Powerim's R&D team under the leadership of Bjørn Arild W. Baugstø is putting behind the implementation of the new SD technology, including the many challenging specification, design and coding tasks that need to be done before this technology can be taken into use. I also want to thank Powerim's partners and key customers for a fruitful cooperation. In particular, I want to mention the people at Statoil, who have supported the project both financially and through their joint visions for an easy-to-use object-oriented modeling tool.

10 References

- Cox, B. (1996). *Superdistribution*, Addison-Wesley Publishing Company, Inc., ISBN 0-201-50208-9.
- Eberlein, R., Hines, J. (1996). *Molecules for modelers*. Proceedings of the International System Dynamics Society. Cambridge: System Dynamics Society.
- Forrester, Jay (1973). *World Model*. Wright-Allen Press, Inc.
- Forrester, J. (1990). *Principles of systems*. Productivity Press, Portland
- Myrtveit, M., Saleh, M. (2000). *Superimposing Dynamic Behavior on Causal Loop Diagrams of System Dynamic Models*. Proceedings of the International System Dynamics Society, Bergen: System Dynamics Society
- Tignor, W., Myrtveit, M. (2000) *Object Oriented Design Patterns and System Dynamics Components*, Proceedings of the International System Dynamics Society, Bergen: System Dynamics Society
- Pollack, Andrew. Two Teams, Two Measures Equaled One Lost Spacecraft. *New York Times*, October 1, 1999
- Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., Nygaard, K. (1973) *Simula Begin*. Chartwell-Bratt Ltd., Bromley, England.

Appendix A—Measurement Units

Simple confusion over whether measurements were metric or not led to the loss of a \$125 million spacecraft last week as it approached Mars, the National Aeronautics and Space Administration said on Thursday. (Pollack 1999)

Variable values are either implicitly or explicitly measured in a unit. Adding explicit units to variables has many advantages. First of all, it becomes easier to detect mistakes like the one that caused the Mars Climate Orbiter to crash. In addition to quality assurance on the model equations, the system can make use of the measurement units when formatting values for output to the user, and also when parsing values from text that has been input by the user.

As an example, the text “10 mph” is an unambiguous specification of a speed, whereas the number 10 alone can be easily misinterpreted when used to express speed.

Measurement units can be divided into two classes, normal units and point units. Point units are used for measurements that are relative to some origin. The most obvious examples are points in time and points on a temperature scale. Obviously, there is a big difference between 3 am and a three hours delay, for example. A delay says nothing about when the delay takes place, only how long it is. Together with a point in time, the delay can be used to find the point in time for the delayed event. Point in time is normally specified as date and time of day. Durations are specified as a number of time intervals, for example years, months, weeks, days, hours, minutes or seconds. The example with temperature is quite similar. What does 2°C mean? Is it a temperature or a temperature difference? If the shower is too cold, we can increase the temperature by 2 degrees, relative to the original temperature. We do not specify the actual temperature—only the incremental change that we want to take place. A particular day it can be

2 degrees outside. In this case we implicitly assume that we talk about a point on the temperature scale, and not a difference between the temperatures of yesterday and today, for example.

Humans are good at implicit interpretations. In a computer model, however, we must be explicit in order to avoid errors. We will use the symbol @ (° can be used as an alternative) to make it explicit when we work with point units. When the point unit indicator (@ or °) is omitted, the normal unit is assumed.

A point unit has an origin, and values are measured as distances from this origin. Values that are measured using point units, behave peculiarly when they appear in mathematical expressions.

Rule	Example
A point value cannot be added to another point value.	DATE(1999, 1, 1)+DATE(2000,1,1) produces an error.
A point value minus a normal value produces a point value	DATE(2000, 1, 3) – 1*DAY() produces a point value (point in time) of January 2, 2000.
A point value plus a normal value produces a point value.	100°C + 10C produces a point value of 110°C.
A point value minus a point value produces a normal value.	DATE(2000,1,4)-DATE(2000,1,1) produces a normal value (time span) of three days.

Modeling software should verify that the above rules are obeyed. In order to do this, we need to be able to define normal units and point units, and assign units to variables.

The format of a unit definition is like this:

```
unit <name> {
  def = <unit expression>
}
```

The unit expression can either be a normal unit expression or a point unit expression.

- **Defining point units**

Point unit expressions are formed using the following syntax:

```
@<unit reference>
°<unit reference>
@(<normal value>, <point value>)
```

A unit reference is either **atomic**, **external**, the name of a predefined unit, the name of a global unit or a dot (.) followed by the name of a local unit:

```
atomic // Used to create a new unit, not compatible with any other unit
external // Used to create a new local unit to be used for formal parameters in
           // order to map to the units of actual paramters when the component is
           // instantiated inside another model
<global unit name> // Definition of named global unit
.<local unit name> // Definition of named local unit
__time // Time
__meter // Length
__kelvin // Temperature
__radian // Angle
__candela // Light
```

```
__kilogram           // Weight
```

A value is a number with unit. A point value has a point unit, and a normal value has a normal unit. In order to define degrees Celsius, we can make the following unit definition, for example:

```
unit C {  
    def = @(1__kelvin, 273.15@__kelvin)  
}
```

The above definition says that C is a point unit that is measured in multiples of one Kelvin from the point 273.15 on the Kelvin scale. Similarly, Fahrenheit can be defined in terms of C, like this:

```
unit F {  
    def = @(5/9C, -(32*5/9)@C)  
}
```

Here F is measured in increments of 5/9 centigrade relative to the temperature $-17,78^{\circ}\text{C}$. An equivalent way to define F is given below:

```
unit F {  
    def = @(5/9__kelvin, 255.37@__kelvin)  
}
```

The time unit is somewhat special. We could of course use the SI unit second to represent time always, but many models count time in other increments, such as years or days. This is why we have introduced the predefined unit **__time**, which is a point unit. A modeling system should define the SI unit second (s) based on the time unit settings for the simulation project. Below is an example, where the TIME variable of the simulation is actually measured in seconds:

```
unit s { def = @(__time, 0@__time) } // Second  
unit min { def = @(60s, 0@s) } // Minute  
unit hr { def = @(60min, 0@min) } // Hour  
unit dy { def = @(24hr, 0@hr) } // Day  
unit wk { def = @(7dy, 0@dy) } // Week
```

- **Defining normal units**

A normal unit expression is defined recursively like this:

```
<unit reference>  
<real>  
(<normal unit expression>)  
<normal unit expression> * <normal unit expression>  
<normal unit expression> / <normal unit expression>  
<normal unit expression> ^ <integer>
```

(The multiplication symbol can be omitted from expressions. **Per** can be used as a substitute for the division operator /.)

Below are some examples:

```
unit m { def = __meter } // Meter  
unit km { def = 1000__meter } // Kilometer  
unit kmh { def = km/hr } // Kilometers per hour  
unit % { def = 0.01 } // Per cent
```

```
unit Acceleration { def = m/s^2 }
```

Appendix B—Syntax Used in Equations

The following is a brief description of the syntax used in the equations of this paper. We use braces ({}) to enclose the body of each definition, as this makes it easier to represent hierarchy. As a convention, each level of nesting is indented one position relative to the enclosing level.

- **Object definition syntax**

The general format of an object definition and its properties is the following:

```
<objecttype> <objectname> {  
  <list of properties>  
}
```

Each entry of the list of properties is either another object definition or a property definition in one of these formats:

```
  <property name> = <value>  
  <property name>
```

Multiple properties on one line must be separated by semicolons (;).

The following shorthand notation can be used for describing a single property:

```
  <property name> <objectname> = <value>  
  <property name> <objectname>
```

Here are two equivalent ways to say that A is equal to B*C:

<u>Object-block syntax</u>	<u>Key-word syntax</u>
<pre>aux A { def = B*C }</pre>	<pre>def A = B*C</pre>

A simulation project has a set of global units and components.

- **Common properties for all objects**

All objects can have one or both of the following common object properties:

```
doc = <text>           // Documentation  
note = <text>         // Note
```

A <text> is a string enclosed in quotes, like this “This is a text”.

- **Units**

A unit is defined like this:

```
unit <name> {  
  <common object properties> // Documentation and/or note  
  def = <unit expression>   // Definition of unit  
}
```

Unit expressions are explained in Appendix A.

- **Variables**

All variables can have any of the following common variable properties:

<common object properties>	// Documentation and/or note
<variable definition list>	// List of local variables
public	// Used if variable is public (excludes private)
private	// Used if variable is private (excludes public)
import = <transfer>	// Defines import type
export = <transfer>	// Defines export type
return	// Return value when invoked as a function
unit = <unit expression>	// Defines unit of measure
dim = <dimensions>	// Defines array dimensions
type = <data type>	// Defines data type (default is real)

The import or export <transfer> can be either **init**, **tail** or **full**. Init means that the initial value is transferred. Tail means that all values, except the initial are transferred. Full means that all values are transferred.

Dimensions and data type are not used in this paper.

A variable can be a reference to another variable. This means that the variable shares the memory with the referenced variable. The **ref** property is used to define a reference.

A <path name> is used to refer to another variable. A path follows the following rules:

<tail of path>	// Path relative to parent of current variable
.<tail of path>	// Path relative to current variable
.. <lt;tail of="" path><="" td=""> <td>// Path relative to grandparent of current variable</td> </lt;tail>	// Path relative to grandparent of current variable
\<tail of path>	// Path relative to toplevel (component)

The definition of <tail of path> is defined recursively like this:

```
<name>
<name>\<tail of path>
```

A variable <expression> follows the normal syntax for mathematical expressions, with the addition of the unit specification that can be added within double angular brackets after a literal value. Here is an example: 10<<miles per hour>>.

A component holds a model and its visualizations (diagrams). The variables of the model are stored inside the root variable, which is of the **mainmodel** variable type.

```
mainmodel <name> {
    <common variable properties> // Zero or more of the common property types
    <unit definition list> // Local units
}
```

A **mainmodel** cannot be a child of other variables.

The unit definition list contains zero or more unit definitions, and the variable definition list consists of any number of variable definitions.

A **mainmodel** also holds properties for defining time and integration settings for the model. These settings are omitted from this overview.

We have the following variable definitions, for the other variable types:

```

level <name> {
    <common variable properties>
    readonly
    ref = <path name>
    init = <expression>
    flows = <flow list>
}

aux <name> {
    <common variable properties>
    ref = <path name>
    def = <expression>
    init = <expression>
}

socket <name> {
    <common variable properties>
    interface = <name>
    ref = <path name>
}

plug <name> {
    <common variable properties>
    interface = <name>
    ref = <path name>
    def = <path name>
}

submodel <name> {
    <common variable properties>
    interface = <name>
    implementation = <name>
}

```

*) Inside a **plug**, the **import/export** settings for variables should be the reverse of what is defined for a **socket** with the same interface name. This way the socket and the plug can be connected (import on one side goes to export on the other, and vice versa).

Ref and **def** are exclusive.

Def and **init** are exclusive.

Appendix C—Summary of Extensions

This paper describes extensions to the modeling language (equations) and the accumulator-flow diagramming language for SD. The number of new and extended features is relatively small. The three main object-oriented features are listed first. The remaining entries are introduced in order to support the introduction of hierarchy, components and wire connections.

The extensions are summarized in the table below. The purposes of each extension are listed as bullet points.

All variable types can hold other variables.

- The main purpose is to be able to create hierarchical models.
- Ability to create local variables to simplify complex definitions.
- Enable components to be called as functions.
- Ability to bundle parameters together inside sockets and plugs for safe and easy component wiring.

All top-level models are components.


- Ability to create building blocks that can be re-used by other components (models).

New **interface** attribute on models, components, sockets and plugs.

- Type-safe and guided connection of and swapping of components without the need to edit equations. (The term swapping describes the process of exchanging one component for another, compatible component. As an example, one provider component can be swapped for another provider implementation, given that they have the same interface.)


New variable type **submodel**.

- Ability to create hierarchical models.
- Ability to create instances of components.


New **submodel**
variable symbol: 

New variable types **socket** and **plug**.

- Bundling of parameters that belong together into one connection point.
- To provide guidance to the user when connecting components together.
- To ensure consistent component connections.
- Remove the need to enter equations as part of the component connection process.


New **socket**
variable symbol: 


New **plug**
variable symbol: 

Wire look on
links and flows
that connect
sockets and plugs
together: 

New reference (**ref**) definition.


- Enable import and export of level variables.
- Increase efficiency of parameter passing (speed and memory).

Reference look
on links: 

Reference
indicator on
variables: 


New variable attributes **public** and **private**.

- Information hiding (between levels of the model hierarchy)

Indicator for
public variables: 

New variable attributes **import** and **export**.

- Define value transfer between submodel and component (when using components as building blocks of other components).

Indicator for
import and
export. Example: 

New variable attributes **inflow** and **outflow**.

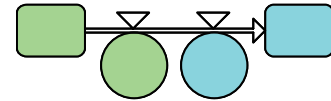
- Allow flows into and out of submodels.
- Prevent flows to be attached to submodels the wrong way.

Indicator for
inflow and
outflow inside
cloud symbol.



Chained flows.

- Passing of flows between sub-models.



New variable attribute **readonly** for levels.

- Control the ability to add new flows to level parameter of submodel.

Various extensions around measurement units.

- Unit consistency checking.
- Automatic unit conversion, for example between meter and inch.
- Flexible component interfaces (through **external** units).
- Formatting of values as text. As an example, an inventory of ten products can be displayed as “10 products” instead of just “10”. Similarly, a value that is a point in time can be shown as “February 2000” instead of “2000.08”.
- Parsing (and conversion) of text input using units and special formats. As an example, a time span can be entered as “1:30:00” and a length can be typed in as “1 inch” or “1/3m”, for example.

Appendix D—Terminology

AFD—Accumulator-Flow Diagram

Accumulator-Flow Diagram—Symbolic language for visualizing models. The language uses a set of symbols for visualizing variables (such as levels and auxiliaries) and another set for visualizing relationships (links and flows)

Auxiliary Variable—Built-in variable type for non-states.

Basic SD—System Dynamics language using only *basic variables*.

Basic Variable—Variable of a built-in type, such as a *level* or an *auxiliary*.

Causal-Loop Diagram—Symbolic language for visualizing *models*. The language uses texts to represent *variables* and *links* to visualize influences between variables.

Child Variable—Variable that exists within a *parent*.

Class—OO term for *implementation* of an *interface*.

CLD—Causal-Loop Diagram

Component—Named entity of a component catalog. A component consists of a model and its visualizations (diagrams). A component can be run both as a stand-alone simulation model and as a submodel of another component. Corresponds to an OO *class*.

Connector—Common name for *socket* and *plug*. Used to connect components using *wires*.

Constant Variable—*Auxiliary* that does not change its value during simulation.

Constructor™ Diagram—Powersim's version of *AFD*.

Encapsulation—*OO* term for information filtering through private *variables* and hidden *implementations* that can be accessed only via *interfaces*.

Extended SD—*Basic SD* with the addition of hierarchy and user-defined variable types, such as *mainmodels*, *submodels*, *sockets*, and *plugs*.

Flow Symbol—Symbol used by *AFD* to visualize flow into or out of *level*.

Implementation—*OO* term for the sections of a *class* that behaves according to the specifications of an *interface*.

Interface—*OO* term for protocol supported by an object.

Level Variable—Variable that holds a state. Changed over time through *flows*.

Link Symbol—Symbol used to visualize relationship between two *variables*.

Mainmodel Variable—Root *variable* of simulation model. Holds other variables.

Model—Simulation model. Contains a *mainmodel* variable with *child* variables.

Object—*OO* term for variable. Objects are of a given *class*.

OO—Object Oriented

Parameter—1) Public *child* of a *parent variable*. 2) Imported or exported child of a *mainmodel*, *submodel*, *socket*, or *plug*.

Parent Variable—*Variable* that has *children*. In *Extended SD* all variable types can have children.

Polymorphism—*OO* term for the fact that one object can take the role of another object, as long as they have the same interface. This can be used, for example, to change a distribution channel variable from direct mail to e-commerce without influencing the structures of the rest of the system.

SD—System Dynamics

Sibling Variables—*Variables* that have the same *parent*.

Socket Variable—*Connector* for passing *parameters* between *submodels* or other *variables*. A socket can be connected to a *plug* with the same interface.

State Variable—Synonym for *level variable*.

Stock—Synonym for *level variable*.

Structured Variable—Variable that has a user defined type. Includes *mainmodel*, *submodel*, *socket* and *plug*.

Submodel Variable—*Variable* type of *Extended SD* for creating hierarchical models and connecting to components.

Thinker™ Diagram—Powersim's version of *CLD*.

Variable—Corresponds to an *OO object*.

Wire Flow—Name of *flow* between *socket* and *plug*.

Wire Link—Name of *link* between *socket* and *plug*.