A framework for dataflow models

August 2013

Magne Myrtveit
Dynaplan As

Abstract

Spreadsheets and system dynamics software are quite different, but they also share some important characteristics: they are both based on dataflow languages. Aspects of dataflow languages are described using a hierarchical framework consisting of five levels: pragmatic, conceptual, semantic, syntax and lexical.

A model is an abstract phenomenon, which needs a representation to be communicated and evaluated. The paper describes two textual representations of dataflow models: one equation based representation suited for people, and one XML based representation suited for software. Both representations are flexible and general enough to represent both spreadsheets and system dynamics models. The relatively fresh modelling software, Dynaplan Smia, is used to illustrate all four levels of the language definition.

The last part of the paper defines the core concepts and features of system dynamics, and shows how extensions can be introduced. An interesting observation is that the core features are sufficient to represent all the models of Sterman's book, *Business Dynamics*, which came out in 2000 and soon became the standard text book within the field.

URL to this paper: http://www.myrtveit.com/papers/A_framework_for_dataflow_models.pdf

Table of contents

1 Introduction to dataflow models

A dataflow model consists mainly of variables, defined using mathematical equations. The value of a variable can be used in the definition of other variables, forming dependencies among the variables. As an example, the following two equations form a dataflow model that calculates the area of a circle with a given radius:

```
radius=10cm
area  of circle=pi*radius^2
```

Figure 1 - Dataflow model calculating area of circle

The term *dataflow* indicates the way values "flow" from independent variables to dependent variables. In our example, *radius* is first calculated, producing the value *10 centimetres*. Once radius is determined, the dependent variable, *area of circle*, can be evaluated. Its right-hand side is a mathematical expression involving one predefined constant (pi), a variable (radius), and a literal (2) grouped together using the operators for multiplication (*) and power (^). The result becomes 3141.5cm^2, which is the result of evaluating π*(10cm)^2.

The most popular dataflow language in use today, is the spreadsheet. Here variables are represented as cells in a tabular grid. Instead of name, which is the common way to refer to a variable in mathematics, variables are referred to by location (row number and cell number) inside a worksheet.

	A	B
1	radius	10
2	area of circle	=PI()*B1^2

Figure 2 – Spreadsheet model calculating area of circle

Equations in a dataflow model can be arranged in any order; it is up to the evaluator to figure out a sequence for calculating the variables in a way that ensures that all variables used on the right-hand side of an equation are evaluated before the equation is evaluated. In our example, this means that *area of circle* must be evaluated after *radius*.

The requirement that the variables on the right-hand side of an equation must be evaluated before the left-hand side is impossible to fulfil if the dependency graph of a dataflow model contains circles. The model below defines population in terms of births, and vice versa.

```
initial population=100persons
population=initial_population + births
births=population*3.7%/year
```

Figure 3 – Circular dataflow model

If typed into a spreadsheet, an error message will be displayed, informing the user that the model cannot be calculated because it contains circular definitions.

The phenomenon of circularity is a good starting point for introducing the concept of feedback. Feedback is caused by delayed interdependencies. In our population example, births cause the population to grow, and after a while new members of the population will eventually get children of their own, causing the population to grow.

Mathematically, feedback can be expressed using integration, like this:

$$P_t = P_0 + \int_0^t B\,dt$$
$$B_t = P_t * 0.037$$
$$P_0 = 100$$

Legend:
P_0 – Initial population
P_t – Population at time *t*
B_t – Birth rate at time *t*
t – Time

Figure 4 – Integral representing population model with feedback

A model involving integration is still a dataflow model. Mathematically, integrals can be very difficult, even impossible, to solve analytically. Fortunately, due to the power of computers, models involving feedback can be computed using numerical integration. The process is called dynamic simulation.

Simulation makes use of the concept of *state variables*, which are used in addition to the ordinary *immediate variables*. During the simulation process, the state variables take on modified values, mimicking how the real system changes its state from one instant to the next.

System dynamics defines methodology as well as simulation technology for working with feedback models. Different vendors have invented their own syntax for expressing their models. Below, is an example how the model in Figure 4 can be expressed in Dynaplan Smia, a modelling language that supports both dynamic and static[1] modelling:

```
person&&persons=unit
initial population&P0=100persons
population&P=stock initial_population inflow births
births&B=population*3.7%/year
horizon=date(2013) to date(2050) step1year
```

Figure 5 – Dynamic dataflow model with feedback

Note: Two built-in units, % and year, are used in the above example, in addition to the user-defined measurement unit, person. Smia supports multiple forms for names, separated by ampersand (&). In the figure, person is defined in singular and plural. P and B are used as short names for population and births, respectively. In Smia, the time dimension of the simulation is determined by the predefined *horizon* type.

When simulated, the model in Figure 5 produces the following development of the stock variable, *population*, displayed as a graph over time:

[1] I use the term *static* for *non-dynamic* dataflow models. Other terms I have considered include *apathic*, *stationary*, *immediate*, and *instantaneous*.

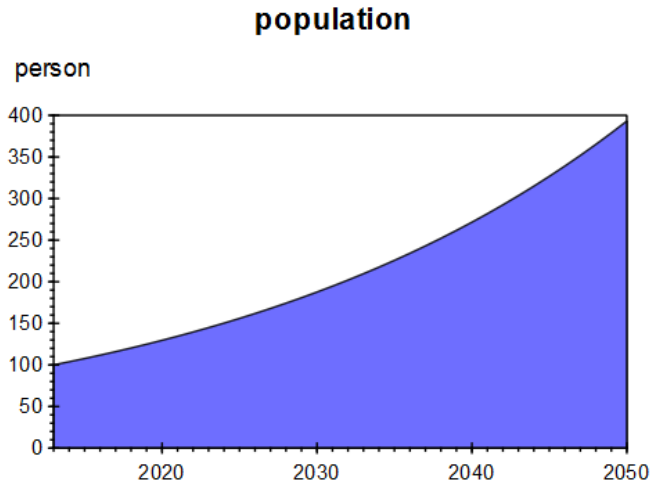

Figure 6 – Development of the population variable over time

Most System dynamics technologies have both a textual and a graphical representation of models. Just as for the equations syntax, also the graphical symbolism varies from vendor to vendor. The stock-and-flow diagram of the population model in our example looks like this in Dynaplan Smia:

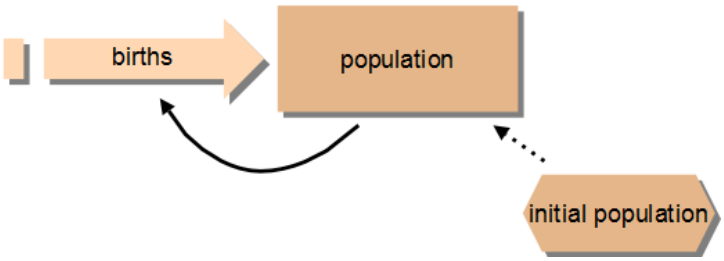

Figure 7 – Stock-and-flow display of population model

2 Background information on Dynaplan Smia

A friend of mine, Lars Vavik, introduced me to dynamic simulation in the early 1980s. It immediately caught my interest, due to its intuitiveness and power to express dynamic phenomena. I used the approach to develop simulation-based applications for education as well as training and process control. Over the years I have been part of the development of four generations of simulation software: SimTek, Constructor, Powersim Studio, and finally: Dynaplan Smia. The focus shifted from application development in the early phase, towards services and software sales later on.

At one point I was quite enthused by the possibility of bringing system dynamics to extensive use in the business world. However, my experience has let me to conclude that the core feature set of system dynamics is useful mainly for education, research, and conceptual work that is not directly linked to the needs of business users. Vendors have to a varying degree realized this as well, as demonstrated by the various extensions that are made for example when it comes to data connectivity and gaming.

Models play a very important role in real life (private, business, government, organisations, education) when it comes to learning, communication, problem solving, planning, entertainment,

training, and much more. The most successful modelling technology is by far the spreadsheet. It puzzles me that it is so popular, in spite of its many well documented weaknesses. While system dynamics is backed by a solid methodology and a very close link to the development of real systems, the spreadsheet comes without methodology and no built-in support for dynamics (feedback).

As mentioned in chapter one, both spreadsheets and system dynamics models are dataflow models. Why is one an enormous success while the other is confined within some relative small niches of the market?

I believe the main reasons are linked to flexibility and ease of use. So I set out to invent a new paradigm for dataflow modelling, combining the main strengths of the spreadsheet with elements from object-orientation, simulation, presentation software, and a few other things. The result is a technology that is system dynamics enabled, without being limited to the system dynamics area of use. Smia has been shown to compete successfully against the spreadsheet not only when it comes to dynamic models, but also in areas where the spreadsheet has its main strengths, such as reporting, consolidation, and budgeting.

To show its versatility, I have implemented import of spreadsheets (Excel) as well as system dynamics models (Vensim) into Smia. To do this, Smia needed a modelling language that is flexible and powerful enough to capture the spreadsheet way of modelling as well as system dynamics.

In Smia, a spreadsheet is treated as a model consisting of worksheets. Each worksheet is a two-dimensional array, with special features:

Array feature	Explanation
Sparse arrays	Arrays can have empty elements (only used elements are stored).
Mixed type array elements	Array elements can have different type, for example text, logical, integer, and real.
Self-referencing arrays	An element of an array can depend directly or indirectly on another element inside the same array.[2]

Figure 8 – Necessary array features for representing a worksheet as array variable

This implies that when a spreadsheet is loaded into Smia, it produces a model with one array variable for each worksheet. The name of the variable is set to the name of the worksheet. (If the spreadsheet contains named ranges or expressions, they too are converted into Smia variables). A syntax inspired by PHP's associative arrays is used for sparse arrays in Smia. Here is the model resulting from importing the spreadsheet in Figure 1 into Smia:

```
Sheet1 = grid {i ≤ 1 to 2,j ≤ 1 to 2
|       1, 1 => "radius"
|       1, 2 => 10
|       2, 1 => "area of circle"
|       2, 2 => pi*B1^2
} as mixed
```

Figure 9 – Smia equation representing worksheet

[2] Vensim supports the criterion of self-referencing arrays, but not the two other criteria.

The keyword **grid** tells Smia that A1 notation is allowed in the formulas when referring to array elements. Braces {...} are used to enclose array definitions. In this example two array dimensions are defined, one named *i* and one named *j*. Both are sparse, as indicated by the ≤ operator, and containing elements in the range 1 through 2. Lines two through fice define the contents of an element (cell) of *Sheet1*. The vertical bar (|) separate individual sparse elements, while the double arrow (=>) separates the key (location) of the element from its definition. Since the array has two sparse dimensions, two indices are listed in the keys section; one for the row and one for the column. The last line (line sixth) closes the array and informs Smia that its elements are of **mixed** type (variants). In our example, we have elements that are texts, integers, and reals.

Note: Importing spreadsheet formulas into Smia is normally not done by users, even if it is possible. Instead, users typically import only the data from their spreadsheets.

Excel does not have an equations language as illustrated in Figure 5. Instead, spreadsheet formulas and data are overlaid inside cells that can be accessed only via the two-dimensional table representing the spreadsheet. In Smia each worksheet has one equation, which is a sparse array variable. The contents of the variable can be viewed and edited within Smia's table object, which behaves exactly like a traditional spreadsheet. This way Smia makes a clear distinction between *model* and *view*[3], something that is missing in current spreadsheets.

Importing a Vensim model, or any other system dynamics model, produces a result that corresponds more directly to the objects inside the source model. As an example, each Vensim variable is converted into a corresponding Smia variable, and each Vensim unit is converted into a corresponding Smia unit. The time specifications in Vensim are imported as variables into Smia, and used to set the definition of Smia's *horizon* type, like this:

```
horizon=Control.'INITIAL TIME' to Control.'FINAL TIME' step Control.'TIME STEP'
```

Figure 10 – Definition of time *horizon* after importing a Vensim model

Due to proprietary, unpublished file formats used by Powersim Studio and isee's iThink[4], I have not been able to implement automatic import and conversions of these formats in Smia. However, I have manually tried to verify that the feature set of Smia is powerful enough to capture most (but not all) of the semantics offered by these tools.

3 Aspects of dataflow languages

The description of a computer language, such a dataflow language, can be organized hierarchically in order to capture different aspects of the language without losing the big picture among all the details. The figure below lists the main levels of a language definition:

```
pragmatic level
conceptual level
semantic and syntactic levels
lexical level
```

Figure 11 – Levels of a language definition

[3] *Model* and *view* are elements of the MVC (model-view-controller) software architecture.
[4] After the first version of Smia was finished, isee has opened up its file format to the public through its support of the emerging XMILE specification.

3.1 Pragmatic level

The *pragmatic level* deals with the purpose of the language. A description of the typical user belongs to this level along with the intended ways the user will make use of the language and its associated technology to fulfil its purpose. A description of the knowledge and experience of users can be organized, for example like this:

education – academic level and disciplines (natural sciences, social sciences, system dynamics, ...)
technologies mastered by users (accounting, spreadsheets, data bases, programming, ...)
industries users work within (automotive, construction, finance, services, education, ...)
job functions users occupy (admin, sales and marketing, production, services, controlling, IT, ...)
geography, cultures, languages, age, etc.

Figure 12 – Framework for knowing the user

The pragmatic level can also contain competitive analyses (advantages and disadvantages compared to alternative solutions) and other market related considerations.

Without a clear understanding of the pragmatic level, it is hard to judge whether or not the modelling software/language is fit for its purpose.

3.2 Conceptual level

To use a modelling language (and its software) correctly and efficiently, users must learn and understand the key *concepts* of the language. It is an advantage if concepts are taken from or anchored to elements of the end users' common knowledge base. The definition of the typical user belongs to the pragmatic level, and provides important inputs to the selection of concepts, conventions, and metaphors that will be easy and intuitive to understand within the target market. If the end-user definition is very broad, it can be a challenge to come up with concepts that work well for everyone.

Concepts found in some (or all) of the dataflow systems I have studied, are listed in the table below:

equations – for defining variables
calculation – for evaluating immediate variables and initializing state variables
simulation – for updating state variables when virtual time is advanced
measurement units – for assigning meaning to numbers
calendars – time and date values
currencies – monetary units and conversion (exchange rates)
hierarchy – abstraction mechanism for managing large models
functions – abstraction mechanism for mapping parameters to outputs
arrays – abstraction mechanism for mapping subscripts to outputs
charts and tables – for visualizing and editing data (and equations)
diagrams – for visualizing variable dependencies (case-and-effect)

Figure 13 – Some concepts that might be used in dataflow models

The list of concepts can be extended almost infinitely, both in depth and breadth. It is provided here mainly as an example. Important concepts, such as, objects (interfaces, classes, instances, inheritance, and polymorphism), stochastic sampling, function sampling, interpolation, extrapolation, data types, sets, name spaces, and lists are omitted for brevity.

To a varying degree, other modelling paradigms make use of different concepts. As an example, *von Neumann* languages (C, C++, C#, FORTRAN, Pascal, Smalltalk, Ada, Java, PHP, Basic, etc.) build on concepts such as the following:

assignment – for setting a variable's value
sequence – for performing an algorithm step by step
branching – for performing different parts of an algorithm based on a condition
iteration – for solving a problem by repeating statements until a certain condition is fulfilled
invocation – delegating control to a sub-algorithm (function)
recursion – invoking a function from within itself if a certain condition is met

Figure 14 – Concepts used in von Neumann style models

Some dataflow concepts are related, but still different, from the concepts above. *Assignment* can be compared to accumulation (inside stocks); *sequence* is determined automatically from the dependencies of a dataflow model; *branching* is local to a single expression in dataflow models; *iteration* is implicitly done by the simulation process, *invocation* and *recursion* are not relevant in dataflow models; however, dataflow models can work with *delegation*, a concept similar to object-oriented concepts in languages derived from von Neumann style of programming.

3.3 Syntactic and semantic levels

The syntax of a language describes the grammar of the language, i.e., how words (tokens) can be combined together into valid sentences (declarations and statements) that can be carried out by a computer (and understood by a human). Along with the syntax goes the meaning, or semantics, of the sentences.

Although syntax and semantics are different aspects of a language, it can be practical to describe them together. One reason for that, is that in order to explain semantics we need a notation for giving examples, which immediately calls for a syntax that we can use.

The semantic language definition also covers what will happen if a "program" (list of sentences) does not obey the grammar or other constraints implied explicitly or implicitly by the language. Instead of crashing or producing the wrong results, the software should provide error messages that can help the user locate and fix the source of the problem.

Grammars differ significantly among computer languages (and spoken languages too), as can be seen from the many variations among existing languages in the dataflow category, the von Neumann category, as well as in languages (LISP, PROLOG, etc) from other categories.

3.3.1 Expressions

For dataflow languages, it is natural to base the grammar on conventional mathematics. This is to a varying degree done by the system dynamics technologies, as well as spreadsheets. The basic grammatical entity borrowed from mathematics, is the *expression*. Expressions can involve literals, operators, functions, and variables. Below are some examples:

1 + 2
∑v
a*f + b*(1-f)
sin(a)/cos(a)

Figure 15 – Examples of mathematical expressions

Special mathematical symbols, such as ∑, are normally spelled out using text when used in computer languages. As an example ∑v might be written as sum(v).

The syntax used above is by no means the only way expressions can be formed. Adobe's Postscript, as an example, uses postfix notation instead of infix, which is used above. In Postscript the examples would become like this:

1 2 **add**
v **sum**
a f **mul** b 1 f **sub mul add**
a **sin** a **cos div**

Figure 16 – Postfix version of mathematical expressions

HP (used to) offer calculators with postfix notation to the scientific community. One advantage of this notation is that there is no need for parenthesis to determine the order in which operators are applied. I do not propose to use postfix – and neither prefix – notation, but it is mentioned here in order to highlight that the same concept (here: mathematical expressions) can be represented using very different grammars (syntax).

Note: Conventional mathematics uses both prefix, infix, and postfix notation, depending on the operator in question. Binary operators use infix, always. Many of the unary operators use prefix syntax, but there are exceptions, such as faculty (!). Trigonometric functions are prefix operators in conventional math, but they are often treated as functions of one parameter in programming languages.

3.3.2 Numeric values

The set of available functions and operators, and the data types that are allowed for literals and values, depend on the scope of the language. At the most basic level (algebra), literals, parameter operands, function parameters, and variable values are restricted to reals.

3.3.3 Units

Numbers normally represent how many units there are of something. As an example, the number 10 can represent the age of a child (number of years), the length of a wall (number of metres), or the number of balls in a bucket. Many computer languages leave the interpretation of the meaning of a number to the user. However, some systems also support the concept of measurement **units** as part of the language definition. This is useful not only in physics, but also in models within other domains. The main advantages of including units in a dataflow language are the following:

readability – The end user immediately sees what a number represents (reduces the chance of misinterpretation).
quality – The system can check that the semantics of units is obeyed in equations, and issue error messages in case incompatible units are involved in addition or subtraction (eliminates the chance of performing illegal operations on units).
expressiveness – The user can freely mix compatible units in expressions, and leave it to the system to perform the necessary conversions (shifts the burden of unit conversion from the modeller to the software, and eliminates the chance of incorrect unit conversions).

Figure 17 – Advantages of measurement units

3.3.4 Logical values

If Boolean algebra is included in the grammar, the **logical** data type must also be added (at least conceptually) to give access to truth values (**true** and **false**). At the lowest level, all data are represented as numbers (bytes) inside the digital computer's memory (RAM). As an example, true can be represented by the number 1 and false by the number 0. Some languages use the number approach, while others provide a separate data type for working with logic.

It is a good design principle to make it the responsibility of the software to perform mappings between real-world (user-level) concepts and their low-level representations. At the modelling language level, a system supporting logical operations must also support the logical data type, in order to be conceptually correct. Only this way the software can ensure that the semantics of logics is not violated by any expression. And only this way the system can contribute to a correct understanding of the concept of formal logic. Both are extra important if the system is used by children, who are developing their understanding of math and logic for the first time.

Correct semantics is a matter of quality (correctness). In addition to that, logical types allow the software to display logical values as *what they are* (true or false) instead of indirectly as *how they are represented* (1 or 0). This enhances readability of models, and brings the system closer to the notion that most users (who are not programmers) have about the nature of logical values.

3.3.5 Text values

To complete the discussion about data types for now, let us assume that the language is extended to include operations on text (strings). Even text is represented as numbers at the lowest level, according to some character encoding (ASCII, Latin1, UTF8, UTF16, or similar). In this case it is very obvious that the system cannot display values in terms of their low-level representation. Try to read this, for example: 116, 101, 120, 116. (It is "TEXT" encoded using the ASCII code table). Text calls for a new data type in our language, the **text**[5] type.

3.3.6 Equations

The expression syntax (and semantics) that is covered above is normally used in the larger context of a variable definition. The mathematical syntax for defining variables is the following:

```
name = expression
```

Figure 18 – Commonly used syntax for variable definitions

The above syntax describes an equation, where the left-hand-side is equal to the expression on the right-hand side. In the general case, the *expression* on the right-hand side can be quite complicated, depending on the grammar of the particular language we are dealing with.

3.3.6.1 Equation types

In case the language includes other objects than just variables, it may be useful to extend the syntax with keyword or other syntax elements that can be used to identify the type of object that is defined by the equation. Below is a list of syntax examples for how a type T with members 1, 2, 3, 4 can be defined:

[5] **Text** is called **string** in many programming languages. But text is a better choice if our user group is not programmers.

```
type T = 1 to 4
T = subset [1,2,3,4]
T = type 1 to 4
T =1 to 4
```

Figure 19 – Syntax examples for defining a type holding the integers 1 through 4

In the first example, the type is identified explicitly by a leading keyword identifying the class of object that is defined. The other examples detect that the defined object must be a type by analyzing the expression at the right-hand side.

For types, it is always possible to determine the equation type from the right-hand side. If unit definitions are supported, however, it is generally not possible to distinguish them from normal variable definition. So, in this case it is necessary with a separate keyword to determine if an equation defines a variable or a unit. See example below:

```
unit km = 1000m
km = unit 1000m
```

Figure 20 – Syntax examples for defining a the unit kilometre (km)

If the **unit** keyword is omitted in the above examples, it seems natural to interpret the definitions as variable definitions. Alternatively, the grammar could deny this by requiring a keyword identifying the equation type also for variables.

The syntax definition of a dataflow language can become quite large, especially if the language is rich on equation types and features. Different vendors / languages have found their own ways to express things, especially in areas where the languages provide unique or innovative features.

3.3.7 Lexical level

Sentences in a textual language are constructed from characters, such as letters, digits, punctuation, spacing, mathematical operators, and other special characters. Characters are grouped together according to language-specific rules to construct the words (tokens) that can be used to construct sentences (equations, etc.).

In a dataflow language, the main token types are the following:

Token type	Examples
literals	12, 123.4, true, "text"
names	price, x, mW, MW, sin, m, 'price list', %, $, €
operators	+ - * / < > = <= >= <> ^ !
parenthesis	( ) { } [ ]
separators	. , ;
other	=> | :

Figure 21 – Sample token used in dataflow models

The set of token types, and the rules for writing them, vary from one language to the other.

4 A language framework for dataflow models

This section starts out with a discussion of a common standard, versus a framework for several co-existing languages. You can skip 4.1 if you are not interested in the discussion, but only in the

framework. The rest of the chapter covers the framework from the pragmatic level down to the lexical level, with examples and some recommendations.

4.1 A common language for dataflow models – pros and cons

From time to time the idea pops up to create a model interchange format (MIF) for transferring models between different languages (technologies). Assuming that such a format was in place, users would be able open models created by someone using a different technology, given that both technologies supported MIF. People have imagined several cases where this could be useful, including the following:

publishing – An author can distribute models in MIF format together with a book, for example on a CD or via the web.
switching technology – A customer has decided to switch technology, and wants to transfer his models to the new platform.
peer review – Reviewers can use their favourite tool to run and analyze models that are part of an academic paper submitted for publishing or presentation.
cooperation – Modellers using different technology can exchange and share their work, fostering learning and communication.
customer power and security – Customers become less dependent on a particular vendor.

Figure 22 – Potential uses of a model interchange format

The "second wave" of the common file format discussion within the system dynamics community has led up to the draft specification of SMILE (a dataflow language for modellers) and XMILE (an XML representation for storing SMILE models together with their graphical visualizatio; diagrams, charts, and tables). The purpose of SMILE has evolved away from model interchange towards model standardisation. In other words, SMILE is intended to become the standard language for system dynamics, and XMILE the standard document format.

Let us assume for a moment that somebody proposed to create a standard language for the von Neumann category of languages. The background for the initiative could be that all the languages in the category build on the same core concepts. In reality, however, the concepts and semantics of languages such as C++ and Java are so different that they cannot be unified without sacrificing one or the other. A common language for von Neumann programming would soon become an additional language, sitting side-by-side with the existing languages. A standard, in the von Neumann world, would mean a selection of one language over the other, and not a unification of the existing languages. Selection takes place naturally through market dynamics, where some languages and their accompanying tools, stand out as better choices than others. A combination of conservatism (communities loyally stick to the tools they are accustomed to) and different requirements in different market segments, creates a situation where different languages can thrive side-by-side, without one wiping out the other.

In my opinion the same situation exists among the system dynamics languages (excluding spreadsheets for a moment, since they are not covered by SMILE). A common standard for system dynamics will become a new language, borrowing concepts and features (to different extents) from existing technologies, and mixing in new ideas. (This is essentially what I did when I started the definition of Dynaplan Smia).

I have several concerns relating to a standardisation of system dynamics on the technology level:

Conservatism – A standard my cast in stone historical, suboptimal solutions that are part of the reason why the SD technologies do not compete more successfully that it has done over the last 40 years.

Fairness – A standard my turn out to be unfair towards vendors who happen to provide features that are not compatible with or included in the standard. Will authors who use a syntax that differs from SMILE have difficulties publishing their work? Will vendors who do not follow SMILE have difficulties staying in the system dynamics market for education, research, and projects?

Innovation – How does new ideas come into the software when they are not part of the standard?

Intellectual property rights

Implementation – Sometimes it is both cheaper and better to tear down a house and build a new one, than it is to modify an existing house to fit new needs. The same might be true for existing system dynamics software if they are going to undergo the changes needed to become SMILE compliant.

Figure 23 – Some concerns about making one system dynamics language the standard

To resolve these issues, it might be best to define a flexible and extendable framework, implement a common core for system dynamics, and letting vendors make extensions within the framework, unrestricted any standards organisation. The paper at hand is my contribution to creating this platform.

4.2 The framework and fragments from an implementation

As part of the definition and development of Dynaplan Smia I needed a framework for defining the language as well as its file format. Especially in the beginning, but also now, Smia is undergoing many changes and extensions. This called for a general and flexible approach to building up and modifying the language, its implementation in software, and its representations (equations and graphics format presented to the users, and file format used when storing documents).

The main features of the framework are captured by the different language levels as introduced in chapter 3, *Aspects of dataflow languages*. In the following the different design decisions are covered in more detail.

4.3 Pragmatic model

The end-user is defined as a non-programmer, and may or may not be a system thinker or trained in system dynamics. No particular region, industry, function, or academic discipline is targeted when designing the system. It is an advantage if the user has an analytical approach to problem solving. The age range of potential users is large, and includes children who learn about modelling in school as well as adults who use modelling in their work.

This means that the language must use concepts, conventions, and metaphors that belong to the common knowledge base of most users, or that can be easily explained and learned.

4.4 Conceptual model

The framework assumes that a dataflow model consists of a collection of objects, arranged in a hierarchy. (The hierarchy is optional if an implementation does not support it). The main properties of an object are its name, definition, and documentation. (Not all object types need to support all three properties. Smia contains object without name, objects without definition, and objects without possibility to have documentation).

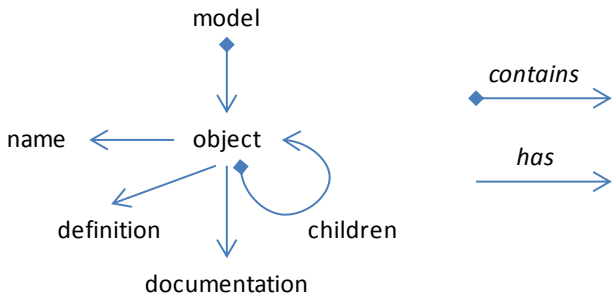

Figure 24 – Structure of a dataflow model

The model is an abstract concept, which needs a representation in order to be viewed and edited by a user. The textual representation of a model is defined by the grammar of the language. In addition, there can be any number of graphical representations, called diagrams, showing some or all of the objects, their properties and relationships to other objects. Diagrams are collected into books, which are contained inside the view part of the dataflow document.

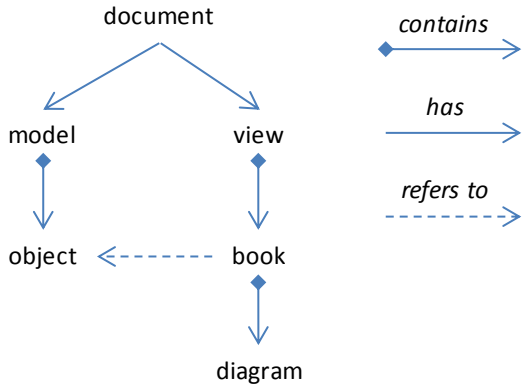

Figure 25 – Structure of a dataflow document

Each diagram book visualizes the model from the perspective of one object, which can be a submodel or a variable, for example. (Implementations that do not support multiple books or multiple diagrams can link the view directly to the diagram object).

Document, model, view and book have additional properties, not covered here. As an example, the document has information about the author of the model, when it was created, etc.

I will leave the view part of a dataflow model open in this document, and concentrate on the abstract model, its visualization as text, and its representation as XML (Extensible Markup Language).

Observe that the above arrangement of objects and object relationships is generic and flexible enough to capture the structure of models created in Vensim, iThink, Powersim, Constuctor, Excel, and of course: Smia.

Before switching to the next topic, syntax and semantics, let us have a closer look at three key concepts of our model definition: naming, hierarchy, and types.

4.4.1 Object naming

A *name* is the most powerful abstraction mechanism there is. It allows us to refer to a phenomenon, small or large, by just one term; its name. Just think about the richness of names such as: “universe”, “atom”, “mother”, ...

A name can be represented in many different ways, for example as a sequence of glyphs (symbols in a font), a combination of sounds (speech), or dots on a surface (Braille). For historical and technical reasons programming languages typically reduce the set of characters can be used in names to the set of Latin characters in combination with digits and possibly some additional characters, such as underscore (_).

Given that our user base is not the programmers of the world, and given that users live in any country or region, object names must be possible to construct from any alphabet, not just Latin.

Different languages and regions use different words when referring to a “thing”. Models that will be used by people from different places on the earth, should allow objects to be named in multiple languages.

Users sometimes use abbreviations for names. This is quite common in mathematics, where names are typically one-letter words (x, y, z). But it is also used in other contexts, especially for objects that are used frequently and where there are established abbreviations that are “known to all”.

Languages have rules for singular and plural name forms. This is typically ignored by programmers, but since our target audience is the “man on the street”, our object naming solution should allow not only for names in different languages, but also according to the rules[6] each language has for distinguishing between one and many.

Some names are defined equally in all languages. This is the case for the SI names for measurements, three-letter codes for currencies, standard codes for countries and languages, etc. Instead of assigning such names to one arbitrarily chosen language (such as Chinese), or copying the names to all languages, there is a better alternative: Define a “neutral language” that is used for names that are independent of any native language. Names in the neutral name are case sensitive. All other names are case insensitive.

In summary, object naming should ideally fulfil the following requirements:

[6] Some languages have different forms for magnitudes of zero, one, two, and even higher number of items denoted by a name. In English we have only two forms, typically distinguished by a terminating ”s” for plural. Whether or not zero is treated as plural or singular depends on the language.

1. Any object has one or more names.
2. A name can be composed from:
    a. Any printable character
    b. Space
    c. Non-breaking space
    d. Soft hyphen
3. Vertical space and tab are not permitted in names.
4. Space, non-breaking space, soft hyphen, and underscore (_) are ignored when comparing names.
5. Each name is associated with a language.
6. A special “neutral language” is used for universally standardized names.
7. Names in the neutral language are case sensitive. Names in all other languages are case insensitive.
8. Every name is associated with a name form, which can be one of the following:
    a. Singular, full name
    b. Singular, abbreviated name
    c. Plural, full name
    d. Plural, abbreviated name
    e. Dummy (Used for computer-generated names, such as 'variable 3' where the name form is not known. If the user adds a name to an object, the dummy name should be deleted automatically).

Figure 26 – Suggested rules for object naming

The distinction between *model* (things) and *view* (visualisation of things) is a key to good language design. As a consequence of applying this principle to naming, TAB, CR and LF are denied as part of object names. It is up to the view part of the software to add indentation and line breaks to a name when displaying it in a given context. Modellers can use the soft hyphen character to give hint to the presentation layer about where to break a name into multiple lines if it does not fit within the available horizontal space.

Some systems support the concept of “alias” names, opening up for synonyms. Aliases are typically used for abbreviations and plural forms. The name forms introduced above (8 a-e) assign a role to a name, allowing the system to pick the appropriate name depending on its context. It also makes it possible to give users the option to decide if they prefer to see long or short versions of names. Short names can be selected when the user is accustomed to the abbreviations used, while long names can be switch on if the user is unsure of the meaning of abbreviations.

A carefully selected, long name can act as an excellent description of a variable, while the short name is a practical way to display the name when space is limited, when the user is accustomed to the name, and when reading speed is important.

4.4.2 Object hierarchy

The hierarchy is maybe the most important abstraction mechanism there is, next to naming. Hierarchies are used to manage complexity by grouping together phenomena, and giving them names. By organising models into an hierarchy, the abstraction mechanism we use in our daily lives can be carried over and used also in the virtual world inside our computer models.

Hierarchy is closely related to naming, as names are typically not globally unique. The term “mother”, for example, is ambiguous. We need to know the mother of whom. In software names

referring into a hierarchy commonly use some form of *path notation*. Some form of path notation is needed if a name can be used for multiple objects in the hierarchy.

The most commonly known path notation is the file path. It consists of a combination of names and the separators colon (:), slash (/) or backslash (\), and period (.). Paths can be relative (starting from a *current location* in the hierarchy, or absolute (starting from a know location in the hierarchy).

It turns out that the semantics of file paths does not work well for dataflow models. Therefore vendors have invented different alternatives for their modelling languages.

Dataflow languages contain a set of predefined named objects in addition to the objects users can make. Historically, programming languages reserve certain names for internal use (syntax). Reserved names cannot be used for user-defined names. For programmers it is Ok to live with the fact that the system has taken away some names for its own use, and that system names cannot be altered. However, the conclusions about who is our user (at the pragmatic level) lead to a different conclusion: Reserved words and functions should be treated as object, residing in the object hierarchy and named according to the rules in Figure 30. When a name is ambiguous, it should be resolved for built-in objects in exactly the same way as for user-defined objects.

Note: In Smia most predefined objects are located inside the predefined submodel *globals*. Users are free to add objects here as well. An absolute name path starts with a #, which refers to the *globals* scope. This makes it very easy to address a global name from anywhere location in the hierarchy.

4.4.3 Types

A type is a concept for defining sets (finite or infinite) of values that can serve as inputs or outputs of models and be part of calculations. There are rules determining valid operations on types and members of types (values). In a dataflow model, types are relevant in several contexts, for example:

Values of variables belong to types.
Function parameters and return values belong to types.
Array dimensions belong to types.
The simulation horizon of a system dynamics model is (at least conceptually) a type.

Figure 27 – Use of types in models

In computer languages, including dataflow languages, the main types are numbers (integer, real), logical, text, and enumerations (lists). In addition some systems support complex numbers, fuzzy numbers, and/or a mixed (variant) type. Real and complex numbers can be with our without measurement units.

4.4.3.1 Special values

If a numerical value is stored using a fixed number of bits, calculations may produce results that are outside the range of values that can be represented. It is also possible to perform operations that involve illegal combinations of inputs, making it mathematically impossible to produce a result. Both situations are accounted for by hardware (or software) when it comes to floating-point representations of real numbers (also called double or float). If sparseness is introduced (a requirement for implementing spreadsheets), the concept of empty is also needed.

Smia includes indefinite, infinity, max, min, and empty as special “values” for every type (where it is relevant). These extensions open up for better quality assurance, and improved conceptual and

computational correctness. The following syntax can (in Smia) be used to specify these special values:

Type	indefinite	empty	min, max, infinity
integer	indefinite(integer)	empty(integer)	min(integer) max(integer) infinity(integer)
real	indefinite(real)	empty(real)	min(real) max(real) infinity(real)
real with unit *m/s*	indefinite(m/s)	empty(m/s)	min(m/s) max(m/s) infinity(m/s)
text	Not available	""	Not available
logical	indefinite(logical)	empty(logical)	min(logical) => false max(logical) => indefinite
list named *n*	indefinite(n)	empty(n)	min(n) max(n) infinity(n)

Figure 28 – Special values for different data types

4.4.3.2 Operations on types

Both for pedagogical (conceptual) reasons and for quality reasons, systems should take great care to ensure correct value mapping within and between types. If a value falls outside its valid input or output domain, the user should be informed.

As often in life, one thing leads to another. The presence of indefinite (nan) and infinities for numerical values, implies that the logical data type must get at least three members: false, true, and indefinite – or in more common terminology: “no”, “yes”, and “don’t know”. Together with support for sparseness (emptiness), this has led to the use of four-state logic in Smia, where logical inputs and outputs can be *false*, *true*, *indefinite*, or *empty*. To see the need for indefinite, just think about the expressions below:

```
x = ln(-2)
p = x<0
q = x>0
r = x=0
```

Figure 29 – Example showing why the need for logical indefinite

The result of the function *ln* is undefined for non-positive inputs. Therefore, the value assigned to x in the first line above, will be *indefinite* (not-a-number). Now, what should be the value of p, q, and r? The answer is that since x is not a number (it is outside the set of valid numbers), it cannot be compared to zero. It simply does not make sense. You might argue that p, q, and r should all be set to *false*. It seems reasonable, but then, have a look at the situation if **not** is put around the comparisons. If x<0 is false, then **not**(x<0) must become true. This means, in turn, that x is greater than or equal to 0, but that is denied by the equations for q and r. So, the only “logical” answer is that p, q, and r all receive the value indefinite (not-a-logical).

Since hardly any existing computer language has taken the full consequence of the above observations, the framework does not require that others conform to a strictly correct definition and

handling of types and values. I include it here to point out something important that people seem not to realize, and as suggestion for further enhancements of existing and emerging software[7].

4.5 Syntactic and semantic levels

Vendors have come up with vastly different solutions to which concepts to support, and how. In the framework I have concentrated on capturing the model as an organisation of potentially nested objects with different properties.

When working with models using software, the graphical user interface provides separate views for editing individual objects and object properties. As an example, the object's name, definition, and documentation are edited inside separate text boxes. The complexity of the grammar is somewhat simplified in this context, as it can concentrate on one definition at the time, without having to deal with object hierarchy and sequence. The formula bar of Excel is an example where focus is restricted to the definition of a single cell inside the scope of the workbook and its different worksheets.

The entire definition of a model displayed as text is useful mainly in situations where the model is published in whole or in part, for example inside a journal or in a text book. The definition of the framework syntax is organized like this:

1. Overall syntax for models. (Representation of the model as a hierarchical collection of objects and properties).
2. Syntax for property values.
3. Syntax for the definition property. (Expressions)
4. Syntax for localisation. (Translation of the model into multiple languages).

Figure 30 – Dataflow syntax levels

4.5.1 Syntax for an object and its properties

The framework contains two interchangeable syntaxes at this level. Each has advantages and disadvantages depending on the situation. Therefore, the user can select which representation to use from case to case.

The following model is used to show the different syntax:

Name	Parent	Definition	Description
model			
rectangle	model	width*height	Area of rectangle
width	rectangle	10m	
height	rectangle	4m	

Figure 31 - Sample model for demonstrating high-level syntax

The figure below illustrates the hierarchy of this model:

[7] In my view, IEEE should add *empty* as a special value for double, and implement its semantics. Hardware and/or software, such as compilers, should support ±infinity, indefinite, and empty also for integers. Corresponding changes should be made to the specification and handling of comparison and branching statements. The **if**-statement in languages such as C++ should have three branches, one for when the condition is true, one for false, and one for indefinite. If the last branch is omitted, an exception should be thrown when the condition is indefinite. I am quite convinced that these changes would catch errors earlier, and make software more robust.

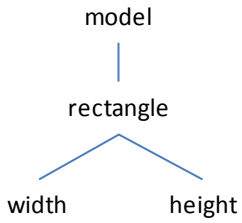

Figure 32 – Hierarchy of sample model

4.5.1.1 Object perspective

In this mode, a model is arranged by object, with its properties nested inside.

```
vendor Dynaplan
product smia
version 4
language enGB
submodel model {
        var rectangle {
                def length*width
                descr Area of the rectangle
→       }
→       var rectangle {
                var length {
                        def 10m
                }
                var width {
                        def 4m
                }
        }
}
```

Figure 33 – Model equations with object perspective

Note: The syntax might be improved by removing the two lines marked with →.

This format is "object-oriented" in the sense that it groups an object and all of its properties together within one sentence block. Braces {...} are used to group objects, and indentation is used to indicate nesting level in order to ease readability[8].

The three lines at the top identify the dataflow language syntax that is used (Dynaplan Smia equations format version 4). The forth line specifies the default language that is used for identifies, in this case British English.

4.5.1.2 Property perspective

In this mode, a model is arranged by property, with objects nested inside.

[8] Powersim Studio uses a similar format for displaying model equations.


```
vendor dynaplan
product smia
version 4
language enGB
def {
        submodel model {
                var rectangle = length*width
→               var rectangle {
                        var length = 10m
                        var width = 4m
                }
        }
}
descr {
        submodel model {
                var rectangle = Area of the rectangle
        }
}
```

Figure 34 – Model equations with property perspective

Note: The syntax might be improved by removing the text “var rectangle” in front of the brace ({) on the line marked with →.

The advantage of this format is that it becomes easy to compare the same property, for example the definition, of multiple objects without being disturbed by other properties, such as documentation.

4.5.1.3 Flat perspectives

An alternative representation of hierarchy is to use absolute path notation for object names, like this:

```
vendor Dynaplan
product smia
version 4
language enGB
submodel model
var model.rectangle {
        def length*width
        descr Area of the rectangle
}
var model.rectangle.length {
        def 10m
}
var model.rectangle.width {
        def 4m
}
```

Figure 35 – Model equations with object perspective, flat version

Here braces are used only to group an object together with its properties.

The corresponding flat property perspective is given below:


```
vendor dynaplan
product smia
version 4
language enGB
def {
        submodel model
        var model.rectangle = length*width
        var model.rectangle.length = 10m
        var model.rectangle.width = 4m
}
descr {
        var model.rectangle = Area of the rectangle
}
```

Figure 36 – Model equations with property perspective, flat version

In this syntax braces are used to group variables having a given property (**def** or **descr**, in this case).

The main advantages of the flat, property-oriented format, is that is uses little horizontal and vertical space, and that objects belonging to different hierarchies can be sorted together.

4.5.2 Syntax for property values

Model objects have properties, such as name, definition, and documentation. The list can be expanded, depending on vendor/language and customer needs. As an example, it can sometimes be useful to include a **value** property holding (a textual representation of) the value resulting from evaluating the model. The **domain**[9] (measurement type and/or unit) of a variable can also be useful to include, either because the language in question requires it, or because it can make it easier to interpret the meaning of the model by inspecting its equations.

The syntax used by the framework is line-oriented. Sentence separators or terminators, such as semicolon (;) are not used, as that would lead to a need for an escape character[10], such as backslash (\), to be used in places where the separator is part of a property value. The line-oriented syntax is easy and intuitive to read by humans, and not difficult to parse by computers. However, when property values span multiple lines, we need to introduce a mechanism for "gluing together" the lines, so the syntax does not become ambiguous[11]. In general, line continuation can be achieved either by putting a "continue on next line" character at the end of the line that is to be continued, or by putting a "continuation of previous line" character at the beginning of the following line. The framework uses the latter approach, for two reasons. First, a line continuation character put at the end of a line would lead to the need for an escape character, which has been argued against above.

[9] I use the term **domain** instead of **unit** for the type and unit property for these reasons. First, **unit** is already used for the unit object type. If we use it also for a property, it will not be possible to do the simplification indicated in Figure 33. Second, the value domain of a variable is determined by its type as well as its unit. Only if the type is real, the **unit** is relevant.

[10] The use of escape characters is second nature for programmers. However, since our typical user is a non-programmer, we try to avoid constructs that favours easy parsing for the software instead of readability for the user.

[11] As an example, a documentation containing a line break followed by the word *var* would cause the parser to treat the line as the beginning of a variable definition as opposed to the continuation of the text. Powersim's equations syntax does not deal with this problem (the syntax definition is ambiguous).

Second, placing the continuation indicator at the following line also makes it possible to use it as a "margin" for property indentation.

```
vendor dynaplan
product smia
version 4
language enGB
def {
    var a = 10.4cm
    var b = 1.32inch
    var bigger = if a ≤ b
    _ then
    _     b
    _ else
    _     a
    _ endif
}
```

Figure 37 – Syntax of multi-line property

Above is an example of a definition property, which spans over multiple lines. It uses an underscore (_) as an indicator that a line is a continuation of the previous one. By convention, the underscore is placed at the indentation level of the property itself, creating a nice separation between the nesting level of the property itself (before the underscore) and the nesting used within the property definition (after the underscore).

Note that when a single property is viewed and edited individually via a widget, line continuation marks are not necessary. As an example, a formula bar for editing the variable *bigger* in our example might look like this:

bigger	=	**if** a ≤ b
		then
		b
		else
		a
		endif

Figure 38 – Line-continuation is not used when individual properties are edited within the GUI

4.5.3 Syntax for the definition property

This property follows the grammar for valid definitions (expressions) for the object type in question. Since vendors define vastly different grammars at this level, its specification is not part of the framework itself. Instead, the framework has a model property holding the name and version of the language that is used for defining equations. From the framework's point of view, a definition can be any sequence of tokens allowed by the lexical level of the language definition. The tokens can be separated by white space both horizontally and vertically. Indentation can be used by the user and/or automatic indentation algorithms implemented by the vendor, to enhance readability of expressions.

4.5.3.1 Syntax and semantics of operators

Since operators make out such an important part of dataflow languages, I will include a suggestion for choice of operators along with their precedence and associativity:

Precedence	Group	Members	Associativity
(highest) 1	Array subscript and name paths.	var() var.var	left to right
2	Prefix functions.	sin x ln x log10 x sqrtpi x and many more...	
3	Postfix functions.	num ! num !!	
4	Power, exponent and square root.	num ^ num matrix ^^ num num² num³ exp x sqrt x	right to left
5	Space operator (multiplication).	num num	left to right
6	Sign, multiplication and division. Per and / are synonyms. Note that A B is the same as A * B, except that A B has higher precedence.	+ num - num num * num matrix ** matrix num / num num per num num divz0 num num divz1 num matrix // matrix matrix \\ matrix int div int int rem int int mod int	
7	Addition, subtraction, and text concatenation (+). Point unit.	val + val val - val num@unit	
8	Comparison.	val < val val <= val val >= val val > val	
9	Equality.	val = val val <> val	
10	Logical not.	not log	
11	Logical and.	log and log	
11 (lowest)	Logical or and exclusive or.	log or log log xor log	

Figure 39 – Suggested operator table

Note: Some operators may need explanation. *Div* is integer division. *Rem* is remainder after division. A point unit is a measurement unit with origin different from zero, such as temperature measured in Celsius or Fahrenheit, or time (which is measured – not from the Big Bang – but from some historical event, such as the birth of Christ).

The precedence of unary plus and minus is determined by the interpretation that -x is the same as (-1)*x. This moves the unary sign operators into the group of multiplication and division.

Multiplication can be expressed using the space operator. The precedence of space is set higher than multiplication and division because of unit expressions. As an example, *products/person month* is interpreted as products/(person*month), while products/person*month is the same as (products/person)*month.

The precedence of power (^), exp, and sqrt are the same, because exp(x) is the same as e^x, and sqrt(x) is the same as x^0.5.

The point unit operator (@ and °) has the same precedence as addition and subtraction. The rationale is that a@b can be written as a*b + 0@b, which is the same as a * scale(b) + origin(b).

The high precedence of prefix functions makes it possible to omit parenthesis unless the parameter is an expression involving lower precedence operators. Study the following examples to get a feeling how expressions involving prefix functions are interpreted:

Expression	Interpretation
sin x + 2	(sin x) + 2
ln 1 / ln 2	(ln 1) / (ln 2)
sin x^2	(sin x)^2
cos 2x	(cos 2) * x

Figure 40 – Sample interpretation of prefix functions

For languages supporting vector and matrix operations, I have made room for multiplication, right and left division, as well as power. The operators are constructed by doubling up the scalar operators like this (**, \\, //, ^^). Using the mathematician's, rather than the programmer's, interpretation of \ is due to the pragmatic discussion about who is the user. (We need no escape anyway, since we use quoting instead).

4.5.3.2 Syntax for arrays

To the extent our language supports arrays we need syntax for expressing array literals as well as array definitions. Braces {...} can serve well as containers for array elements[12].

4.5.3.3 Syntax for lists (enumerations)

If our language supports element lists, brackets [...] might serve well to enclose the elements. Some languages use brackets for array subscripts. In my view, parenthesis serves just as well. An array is just a special kind of function, anyway, defining a mapping from inputs to values. The expression f(i) is a function invocation if *f* is a function and an array lookup if *f* is an array variable. Since there is really a lack of good grouping characters on the character set, the use of brackets should be selected by great care.

In Smia, brackets are also used for defining subsets of any type. This comes quite natural, since a list of names it not very different from a list of numbers. An example of a numeric subset is the following: numbers = [1, 3, 9, 13].

[12] Using braces for inserting comments into equations "wastes" powerful characters that should be used for more important purposes.

4.5.4 Syntax for localisation

The dataflow framework supports object naming in any number of languages. When working with a model, there is one active language, which takes priority when displaying names, and which becomes of the language of new names given to objects. Below, is a version of the model in Figure 37 (repeated to the right), after Norwegian Nynorsk has been set as the current language, and the three variables have been translated into that language.

Multi-lingual version, with Norwegian preferred	Original English version

vendor dynaplan
product smia
version 4
language nnNO
def {
  var andre = 1,32tomme
  var første = 10,4cm
  var størst = **dersom** første ≤ andre
  _ **så**
  _ andre
  _ **elles**
  _ første
  _ **sluttdersom**
}
trans enGB **name** {
  var andre = b
  var første = a
  var størst = bigger
}
	vendor dynaplan
	product smia
	version 4
	language enGB
	def {
	  var a = 10.4cm
	  var b = 1.32inch
	  var bigger = **if** a ≤ b
	  _ **then**
	  _ b
	  _ **else**
	  _ a
	  _ **endif**
	}

Figure 41 – Syntax of multi-lingual model

Note: The keywords (vendor, product, version, etc) of the syntax are not language-dependant. Maybe they should?

When a model contains names in multiple languages, objects will be displayed with the name in the currently active language (which in our example is Nynorsk to the left and English to the right). Note that user-defined names as well as built-in names of keyword and functions are displayed in the active language. Literals, such as 1.32, are displayed using the decimal separator used by the region selected by the **language** property. Norwegian uses comma (,) while British English uses period (.) as decimal separator.

A model contains one **trans** block for every language, except for the active language. Here object names are mapped from the active language to the language in question. In our example, there is a trans block for enGB, mapping the name property for every object from its nnNO version to the enGB counterpart. Names for built-in objects are obtained from an external language file.

The documentation of objects can be written in multiple languages as well. In this case, a **trans** block for the **descr** property will be added.

If an object is not named in the currently active name, another name will be selected among the available translations. The list of languages will be searched according to the preferences set by the user. A language prefix will be added to the name in order to determine its language.

Original model, written in Nynorsk	English added, but only ”to” is translated
vendor dynaplan	**vendor** dynaplan
product smia	**product** smia
version 4	**version** 4
language nnNO	**language** enGB
def {	**def** {
var ein = 1	**var** _nn_ein = 1
var to = 2	**var** _nn_tre = 3
var tre = 3	**var** two = 2
}	}
	trans nnNO **name** {
	var two = to
	}

Figure 42 – Untranslated (left) and partially translated model (right)

In the above example (right), *_nn_* is put in front of names that are not yet defined in the active language, which is English. Since the variable named *to* in Norwegian has been translated into English (two), the name needs no prefix. It is, on the other hand, listed in the **trans** section for Norwegian.

4.5.5 Summary of syntax

The dataflow syntax framework for models is a line-oriented format, aiming at readability for the end-user, while at the same time being possible to parse unambiguously by software. The format is defined in a way that allows object properties to consist of any text, spanning any number of lines.

The syntax has six different line formats:

Structure of line	Examples	Purpose
prop value	**def** a+b	Assign *value* to property.
prop {	**def** {	Enter scope for defining the named *prop* for the objects listed inside the scope.
class name = value	**var** x=a+b	Define object of given *class*, make it a child of the object defined at the current scope, set its *name*, and assign *value* to the property determined by the enclosing scope (see above).
class name {	**var** x {	Start definition of object of given *class*, make it a child of the object defined at the current scope, set its *name*, and enter scope.
trans *lan prop* {	**trans** enGB **name** {	Enter scope for translating the property *prop* into the language *lan*.
}	}	Leave scope.
_	_	Continue previous line.

Figure 43 – Syntax of multi-line property

White space can be used freely in the front of any line.

4.6 Lexical level

The framework does not define the grammar for tokens used when building sentences in a dataflow language. However, if someone considers developing a dataflow language, or enhancing an existing language by introducing new token types, here are some suggestions.

It is a quite elaborate task to come up with a good choice of characters for representing tokens and special names that are needed in defining the grammar of a language. It is impossible to do a good job without having all the pieces of the puzzle together at the same time, and performing an iterative process of assigning and reassigning characters and character sequences to different purposes until a satisfactory result is achieved. As always, it is wise to use existing conventions within the user community whenever possible, but sometimes there are conflicting conventions used within different communities (mathematics, programming, physics, locations, etc).

4.6.1 Names (identifiers)

A name is a sequence of characters, preferably following the rules in Figure 26. When a name occurs in a larger context, such as an expression, it is necessary to be able to determine where the name starts and where it ends. Software languages use several approaches for doing this. On approach, used by Vensim, is that reserved[13] words and special symbols are identified first, and anything between is treated as identifiers, numbers, or white space. Another, more common approach is to define a grammar for identifiers in a way that they never will come in conflict with neighbouring syntax elements. The most popular approach is to define an identifier as a letter followed by any combination or letters and digits, for example like this: BANK1. In our case it is too strict to limit names like that; we need to be able to include spaces and other characters in names. There are two obvious approaches, at least to a programmer; either use quotes or use escape characters when needed. Of the two, quotes are by far the most easy and intuitive approach for our user base, so we should go for that.

Not only names, but also texts, need to be quoted. We cannot use the same quote character for both, since a text and a name are two very different things in the grammatical sense. Based on conventions established for written text (literature) as well as most existing programming languages, the double quote ("...") is reserved for text, while the apostrophe ('...') is used for names.

In order to allow for the quote character inside a quoted name, two quotes are written for every quote in the name. As an example the text *boy's* is quoted like this: 'boy''s'.

4.6.1.1 Single-character names

Some characters represent names on their own. Examples are given below:

Character class	Examples
Currencies	$ ¢ £ ¤ ¥ ֏ ₡ ₲ ₣ ₤ ₥ ₦ ₧ Rs ₩ ₫ €
Units	% ‰

Figure 44 – Single-character names

The characters above should be interpreted as valid names, and not require quotes. They all belong to neutral language. If single-character names are part of an identifier, the identifier must be quoted.

4.6.1.2 Special names

Languages make use of parenthesis, separators, and other non-alphanumeric characters to form tokens that can be used to construct sentences. Special names consist of sequences of special characters, and do not require quotes. Here are some common examples:

[13] In Vensim reserved words are enclosed between colons. :AND: is an example.

Special name	Use
(* and *) /* and */	Enclosing comments. A comment together with its enclosing text is normally not treated as a token, but considered part of the white-space between tokens. (Depends on software)
+, -, *, /, \, ^, <, <=, =, >=, >, =, <>, !=, ≠, ≤, ≥, !, !!, ++, --, **, //, \\	Mathematical and relational operators. (Depends on software)
, and ;	List separators. (Depends on software, region, and language)
(and) [and] { and }	Nesting of structures, grouping of expressions, function parameter lists, array subscripts, enclosing comments. (Depends on software)

Figure 45 – Examples of special names

The need for, and the use of, special symbols vary among languages. Special symbols normally belong to the neutral language, but there are exceptions. The list separator object, for example, is a comma (,) in some languages and a dot (.) in other.

4.6.1.3 Name forms

If multiple forms of names are supported, as suggested in Figure 26, syntax is required for specifying and separating the forms in the place where an object is given a name. My proposal is to list the different names in a predefined order, specified by item 8, a-e in the mentioned figure. As separator, ampersand (&) can be used. Below is an example of how the unit kilometre can be defined using this approach:

```
kilometre&km&kilometres = 1000 metre
```

Figure 46 – Specifying multiple forms of a name

Name forms can be omitted from right to left, and they can be left empty if identical to an earlier, corresponding form. If short plural is not given, for example, short singular is used. If short singular is missing, long singular is used. If long plural is missing, long singular is used. At least one name form must be specified, but it need not be the first one.

If an object is created with a dummy name, generated by the system, its name will become something like this: *&&&&variable 1*. As mentioned before, dummy name forms are auto-deleted when an object gets a real name.

A system, which does not support name forms, just uses one form and no form separator (&).

4.6.1.4 Language specification

When more than one language is present in a model, a name may need to be associated with the language it belongs to. It is important that the language specification is lean (does not take up much space) and clear (easy to distinguish by humans as well as the language parser). I have landed on a solution where language is put as an optional prefix to name(s), enclosed by underscore (_). Here is an example:

```
fruit = [ apple&_nn_eple, banana&_nn_banan, orange&_nn_appelsin]
```

Figure 47 – Specifying multiple translations of a name

Again the & character is used to separate name forms. The prefix _nn_ used on three of the forms indicate that they belong to Nynorsk-Norwegian. When the prefix is omitted, the name belongs to the active language, which is this case is English.

Two consecutive underscore characters are used for neutral language.

Note that language prefix is rarely needed when modelling. But it can be necessary, for example in situations where the same text is used to mean different things in different languages. As an example, *to* in English can be interpreted as an abbreviation for *torsdag* (Thrusday) in Norwegian. And the name *time* in English, means *hour* in Norwegian, so that must be quoted too if the modeller wants to refer to the English object called time in a context where Norwegian is active language.

Another way to supporting multiple languages is to replace the symbol table when switching language, as opposed to switching the active language. The first approach is used by Excel, for example. It has the major drawback that users cannot use their knowledge about the name of an objects in another language when editing the model. A typical situation is that a user edits a model in his native language (French, say), and he needs to enter the name of a function that he knows the English name for, but he has forgotten what it is called in the active language. With Excel this can be quite painful (I am speaking out of experience with my Norwegian MS Office), while it in Smia is no big deal. The Smia user can just type in _en_ followed by the function name in English, and voila: Smia recognizes the English name and translates it immediately into the active tongue (if available).

4.6.1.5 Path specification

If hierarchical models are supported, and the system does not ensure that all names are unique within the entire model, path notation is required. Many different solutions exist, but to my knowledge, no established software as found a user-friendly and flexible solution for dataflow models. (The solutions do work, but in my opinion, they do not work *well*).

The main objective behind Dynaplan's path notation is to reflect day-to-day conventions for name resolution, rather than strict navigation of a static hierarchy, suited for programmers. In the real world, if a name is unique, it can be used without closer explanation. This is also true for the path notation recommended here. When a name is not globally unique, it can still be possible to locate the object which is referred to, by searching the hierarchy according to certain rules. When this is not possible, the path notation gives the modeller very easy access to the most frequently used locations for initiating a name search:

empty	– look among my siblings
dot (.)	– look among my children
hash (#)	– look among the global objects

The table below gives a more detailed summary, and some examples, of the path notation used by Smia:

Syntax	Examples	Referred object
name	x	If my parent has an object called x, refer to that object. Otherwise look for x in the hierarchy, in a certain order. If x is found only in one place, refer to the associated object. If x unambiguously identified based on other rules, use the corresponding object. Otherwise, treat x as undefined or ambiguous.
#	#	*globals*
#name	#m	Global object named 'm', which is the predefined unit metre.
.	.	Me, i.e., the current object.
..	..	My parent. Each new dot moves one level up the hierarchy.
.name	.y	My child named y.
..name	..x	My parent's child x, i.e., my sibling named x (or myself if my name is x).
name.name	x.y	Child y of object x.

Figure 48 – Path notation used by Smia

In summary, a relative path starts with the *parent* of the current object. A hash character (#) is used to start an absolute path, which begins with the *globals*[14] object. A path starting with a dot (.) begins with the current object. A name in the path refers to a child of the object identified so far along the path. Each successive dot in a path moves one level towards the root of the hierarchy. Any number of dots and any number of repetitions of *.name* can occur in a path, as long as the path at no point refers to the parent of the root, or a non-existing child.

4.6.2 Literals

A literal denotes a given value – a member of a type. Sometimes the term constant is also used, but that is better reserved for a *named literal*, such as the speed of light (c) and the acceleration of gravity (g).

There are established conventions for many types of literals, but not all. Below are some recommendations.

4.6.2.1 Integers

An integer is represented as one or more digits, for example like this: 1234

4.6.2.2 Reals

A real is represented either using scientific notation or decimal notation. Group separators can be used when showing real values as output to the user, but the group separator is not allowed inside expressions. A decimal separator is used between the integer and the decimal part of a real

[14] In Smia, the absolute path does not start with the root, but with a child of the root, called *globals*. Predefined objects are put inside globals, leaving the root level free for the modeller to use without having to worry about name conflicts system objects (there can be hundreds). Starting the absolute path with globals has the advantage that a system defined name, or any other name the user wants easy access to from anywhere in the model, can be accessed just by putting a # in front of the name.

numbers. Choice of group separator and decimal separator depends on the active language in the model.

In my view, the integer part should never be empty. I know this is common in the programming community but it is just laziness and makes equations harder to read. If path notation similar to Figure 48 is implemented, a dot (.) represents the current object. It is possible to make the exception that if a dot is followed by a digit, then it is part of a number. But why bother, when it is so easy to make the definition of a real clearer, and closer to the common user's view of what a number looks like?

4.6.2.3 Texts (strings)

A text string is enclosed in quotes, like this "...". Strings are not allowed to span multiple lines. A paragraph (§) is used to place line-breaks into a text. When displayed to the user, for example as the output of an operation, the text will be broken into a new line where § is found. If a quote (") or a paragraph (§) is to be part of a text, they must be doubled.

4.6.2.4 Logicals (Booleans) and list elements

Logical values are represented by name, in the model language that is active. In English, *false* and *true* are used.

If the dataflow language supports enumerations (lists), each element is displayed using its name in the active model language.

5 A file format for dataflow models

This chapter starts with a discussion of different ways to organise information in a file. A multi-part binary document structure is suggested as the best option for dataflow models. Parts can be equations, views, pictures, texts, etc. Each part is stored in a format that is suitable for that part. Many of the parts are represented using XML. Towards the end of the chapter an XML representation of the model equations part is presented.

5.1 Choice of file format

Models are stored in the file system mainly for persistence (so the application can be closed down when not working with a model), backup, and transfer (copying, mailing, etc). Additional considerations are also relevant when defining a file format for models:

Consideration	Comments
Size and compression	Useful for operational business models that typically contain scenarios with lots of data.
Encryption	Useful if models contain confidential information.
Speed	Speed of save and load is mainly an issue with very large models.
Flexibility and generality	Is the format flexible and general enough to serve as a common file format for all kinds of dataflow models? How easy is it to create new versions of the file format when model features are added, removed or changed?

Figure 49 – Considerations relating to choice of file format

Models consist of multiple parts (equations, diagrams, pictures, etc) which can be relevant for different applications or different uses of the model:

Application	Operation	Relevant file parts
Explorer	Properties	Information about author, application, and model summary information
Explorer	Preview	Thumbnail picture
Application	Edit model	Entire file
Application	Load as library	Equations part is required. Other parts are optional.
Simulation server	Run model	Only the equations are needed. Views are not used.
Presentation server	Control model	Individual view parts and associated pictures, texts and shared formatting properties are needed.

Figure 50 – Different applications and operations require access to different parts of a model

A number of different approaches can be used to organize the parts of a document:

Approach	Description
Files inside directory	The document is represented as a folder, and the parts as individual files inside the parent folder or sub folders of the parent folder.
Multi-part binary file	Here the file contains a directory part that is used to locate different other file parts inside the file. The file itself is a binary file, and parts are accessed using random access file operations. Examples include Windows Compound Binary File Format and ZIP compressed folders.
Single-part binary file	The document is stored as a binary file without directory information. Parts are typically represented as blocks of data, with a header identifying the block type and its size in bytes. IFF is an example of a public structuring standard for binary data.
Text file	The document is stored as a sequence of characters. A meta language is defined for identifying the structure of the file into parts. RTF and XML are examples of public meta language definitions for structuring data as text.
Database	The document is stored in a database managed by a database server. Examples include mySql, DB2, MS Access and MS Sql.

Figure 51 – Approaches to organizing multi-part documents

Existing dataflow systems use different approached among those listed above:

Software	Structuring of parts	Representation of parts	Comment
Excel v.2 and 3	Single-part binary file	Blocks	
Excel v.4	Single-part binary file	Blocks	Adds support for multiple sheets
Excel v.5	Windows Compound Binary File Format	Blocks	(This is a multi-part binary format).
Excel v.7 (Excel 1995)	Windows Compound Binary File Format	Blocks	(This is a multi-part binary format).
Excel v.8 (Excel 1997, 2000, XP)	Windows Compound Binary File Format	Blocks	Adds Unicode support. (This is a multi-part binary format).
Excel 2007 and later, Dynaplan Smia	ZIP (compressed folder) format	Format depends on type of part. XML is heavily used.	(This is a multi-part binary format).
Powersim Constructor and Studio	IFF	IFF	Format is not published by vendor. (This is a single-part binary format).
Vensim	Text	Text	In addition, Vensim has an unpublished binary format.
iThink	Text (combined with separate files)	XML	New format based on draft XMILE specification.

Figure 52 – Current vendor's approach to storing their multi-part documents

The formats have different strengths and weaknesses:

Criteria	Files inside directory	Multi-part binary file	Single-part binary file	Text file
Compression of individual parts	yes	yes	-	-
Compression of whole file	-	yes	yes	yes
Encryption of individual parts	yes	yes	-	-
Encryption of whole file	-	yes	yes	yes
Support for parts with different content types	yes	yes	-	[15]
Speed searching and loading individual parts	fast	very fast	medium	slow
Coupling of parts	poor	high	high	high

Figure 53 – Comparison of the approaches for structuring files

The multi-part binary file stands out as the best approach for string multi-part documents. The approach is used by Excel since version 5, and by the entire Microsoft office suite since 2007. Dynaplan Smia uses the ZIP format as its basis, heavily inspired by Microsoft's solutions, which are now published and readily available for those who are interested.

The figure below provides overview of main parts of a Smia file, and how they are organized.

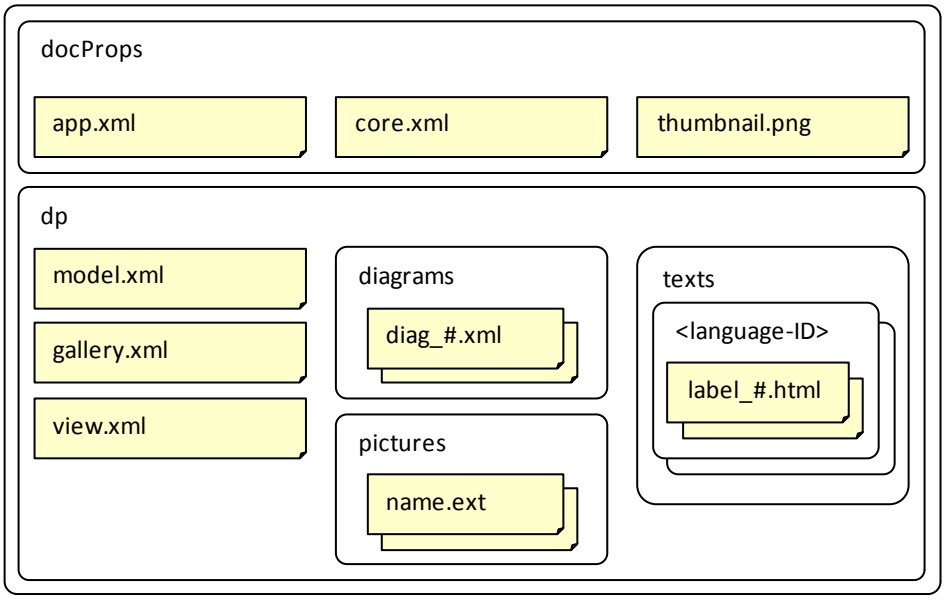

Figure 54 – Structure of Dynaplan Smia DPAX file

The number character (#) is a placeholder for the ID of the owning object inside the model.

[15] In XML Data URLs can be used to embed parts in non-XML format inside a document. A similar approach can be used by other text-based formats.

Yellow boxes represent parts (internal files) while white boxes represent containment (internal directories).

Most parts are represented as XML, pointing at DTD (Document Type Definition) and XSD (XML Schema Definition) specifications stored by the respective owners of the formats that are used. Some parts are in other formats than XML, for example the thumbnail and pictures collection (which are bitmaps) and texts (which are in HTML).

The docProps section follows the specification from Microsoft, providing information about the application, the user, and a preview picture of the document. An example of the app part is given below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Properties xmlns="http://schemas.openxmlformats.org/officeDocument/2006/extended-properties">
  <Application>Dynaplan Smia</Application>
  <Company>Dynaplan As</Company>
  <AppVersion>1.1</AppVersion>
</Properties>
```

Figure 55 – Sample contents of docProps/app.xml

Information about the author and the editing history of the document is stored in the core part:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<cp:coreProperties xmlns:cp="http://schemas.openxmlformats.org/package/2006/metadata/core-
properties" xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:dcterms="http://purl.org/dc/terms/"
xmlns:dcmitype="http://purl.org/dc/dcmitype/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <dc:creator>Magne Myrtveit</dc:creator>
  <cp:lastModifiedBy>Magne Myrtveit</cp:lastModifiedBy>
  <dcterms:created xsi:type="dcterms:W3CDTF">2013-07-31T16:50:07</dcterms:created>
  <dcterms:modified xsi:type="dcterms:W3CDTF">2013-08-01T12:46:00</dcterms:modified>
</cp:coreProperties>
```

Figure 56 – Sample contents of docProps/core.xml

The gallery part contains information that is used by the view part when displaying models. The part will not be covered here.

The view part represents the visualisation(s) of the model. It will not be covered here.

5.2 Model equations represented as XML

The model.xml part of a file is represented as XML[16]. Below a (somewhat simplified) example of the XML that is generated when saving a Smia document is presented. The model is the one that was introduced in Figure 31, page 20. It is repeated below, for easier comparison between the language syntax and the XML syntax.

[16] In addition to XML Smia provides a faster and more compact binary representation of the model part.

```
vendor Dynaplan
product smia
version 4
language enGB
submodel model {
        var rectangle {
                def length*width
                descr Area of the rectangle
        }
        var rectangle {
                var length {
                        def 10m
                }
                var width {
                        def 4m
                }
        }
}
```

The corresponding XML, which can be downloaded from http://www.myrtveit.com/models/rectangle.xml, looks like this:

```
 1  <?xml version=”1.0” encoding=”UTF-8” standalone="no"?>
 2  <!DOCTYPE m:Model SYSTEM "http://www.myrtveit.com/schemas/model.dtd">
 3  <m:Model xmlns:m="http://www.myrtveit.com/dataflowml.model"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xlink="http://www.w3.org/1999/xlink"
          xsi:schemaLocation="http://www.myrtveit.com/dataflowml.model
          http://www.myrtveit.com/schemas/model.xsd">
 4    <m:Properties/>
 5    <m:Locales Current="en-GB">
 6      <m:Locale LocaleName="de-CH"/>
 7    </m:Locales>
 8    <m:Submodel Id="obj-1">
 9      <m:Names>
10        <m:Name>model</m:Name>
11        <m:Name Locale="de-CH">Modell</m:Name>
12      </m:Names>
13      <m:Variable Id="obj-253" Definition="length*width">
14        <m:Names>
15          <m:Name>rectangle</m:Name>
16          <m:Name Locale="de-CH">Rechteck</m:Name>
17        </m:Names>
18        <m:Description>
19          <m:TextDocument Format="xhtml" Locale="en-GB" xlink:href="dp/texts/en-
          GB/descr_253.html"/>
20        </m:Description>
21        <m:Variable Id="obj-261" Definition="4m">
22          <m:Names>
23            <m:Name>width</m:Name>
24            <m:Name Locale="de-CH">Breite</m:Name>
25          </m:Names>
26        </m:Variable>
```



```
27          <m:Variable Id="obj-259" Definition="4m">
28            <m:Names>
29              <m:Name>length</m:Name>
30              <m:Name Locale="de-CH">Länge</m:Name>
31            </m:Names>
32          </m:Variable>
33        </m:Variable>
34      </m:Submodel>
35  </m:Model>
```

Figure 57 – XML representation of a model

Here is an explanation to some of the elements.

Lines	Element	Explanation
4	m:Properties	Model properties go here, such as choice of integration method.
5-7	m:Locales	The active translation is determined by the Current attribute. Additional translations are listed as child elements, if present.
6	m:Locale	Indication that the model is also translated into Swiss German.
8-34	m:Submodel	Element representing the topmost submodel object, named *model*. A required attribute holds the unique ID of the object.
9-12	m:Names	Element holding all the translations of the submodel's name.
10	m:Name	Name in active locale, as set on line 5.
11	m:Name	Name in Swiss German.
13-33	m:Variable	Element representing *rectangle* variable. It definition is provided by the *Definition* property.
14-17	(Similar to 9-12)	
18-20	m: Description	Element representing the description of the *rectangle* variable.
19	m:TextDocument	Element representing the British English translation of the description. If more translations exist they will be added here. Note that the text itself is stored in a separate "part", referred to by the xlink:href attribute. In a single-part file, a data URL could be used instead.
21-26	(Similar to 13-17)	Element defining variable *width*.
27-32	(Similar to 13-17)	Element defining variable *length*.

Figure 58 – Model XML explained

Dynaplan Smia provides a complete DTD and XSD specification of the XML used when storing files.

5.3 Framework DTD for dataflow XML model

While being more detailed, the size and readability of XSD makes it quite difficult to read for humans, unless it is mapped to a graphical representation using a tool such as Altova XMLSpy.

The DTD, on the other hand, is relatively readable. It also has the advantage that an XML file can be validated against its DTD using various tools, such as the one mentioned above. The following DTD can be downloaded from http://www.myrtveit.com/schemas/model.dtd. (The corresponding XSD is located at http://www.myrtveit.com/schemas/model.xsd).

```
<!-- © 2009, Magne Myrtveit, Dynaplan As – Framewok DTD for dataflow model -->
<!—Location: http://www.myrtveit.com/schemas/model.dtd -->

<!-- Format used by text document -->
<!NOTATION xhtml SYSTEM "text/xhtml">
```

```
<!-- Root of dataflow model -->
<!ELEMENT m:Model
        (       m:Properties?
        ,       m:Locales
        ,       m:Submodel+
        )
>
<!ATTLIST m:Model
        xmlns:m CDATA #FIXED "http://www.myrtveit.com/dataflowml.model"
        xmlns CDATA #FIXED "http://www. myrtveit.com/dataflowml.model"
        xmlns:t CDATA #FIXED "http://www. myrtveit.com/types"
        xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink"
        xmlns:xsd CDATA #FIXED "http://www.w3.org/2001/XMLSchema"
>

<!-- Locales (language and country) for names used by model -->
<!ELEMENT m:Locales (m:Locale*)>
<!ATTLIST m:Locales
        Current CDATA #IMPLIED
>

<!-- Locale used by model -->
<!ELEMENT m:Locale EMPTY>
<!ATTLIST m:Locale
        LocaleName CDATA #IMPLIED
>

<!-- Properties of model -->
<!ELEMENT m:Properties
        (       m:UncertaintySettings?
        ,       m:SimulationSettings?
        )
>
<!ATTLIST m:Properties
        AutoEvaluate CDATA #IMPLIED
        CellRefMode (ModeA1 | ModeR1C1) #IMPLIED
>

<!-- Settings for uncertainty -->
<!ELEMENT m:UncertaintySettings EMPTY>
<!ATTLIST m:UncertaintySettings
        Mode (MeanValue | RandomValue | IdealDistribution | RandomDistribution) #IMPLIED
        MaxRuns CDATA #IMPLIED
        MaxSamplesPerVariable CDATA #IMPLIED
        FixInitialSeed CDATA #IMPLIED
        InitialSeed CDATA #IMPLIED
>

<!-- Settings for simulation -->
<!ELEMENT m:SimulationSettings EMPTY>
<!ATTLIST m:SimulationSettings
        UpdateViewsDuringSimulation CDATA #IMPLIED
        Integrator (Euler | RK2 | RK3 | RK4) #IMPLIED
>

<!-- Settings for run-time checks -->
<!ELEMENT m:RuntimeChecks EMPTY>
```



```
<!ATTLIST m:RuntimeChecks
        Mode (Report | Silent | Off) #IMPLIED
>

<!-- Submodel object -->
<!ELEMENT m:Submodel
        (       m:Names
        ,       m:Description?
        ,       (       m:Variable
                |       m:Unit
                |       m:Type
                |       m:Submodel
                )*
        )
>
<!ATTLIST m:Submodel
        Id ID #REQUIRED
>

<!-- Variable object -->
<!ELEMENT m:Variable
        (       m:Names
        ,       m:Description?
        ,       (       m:Variable
                |       m:Submodel
                |       m:Index
                )*
        )
>
<!ATTLIST m:Variable
        Id ID #REQUIRED
        Definition CDATA #IMPLIED
>

<!-- Type object -->
<!ELEMENT m:Type
        (       m:Names
        ,       m:Description?
        ,       (       m:Type
                |       m:Element
                )*
        )
>
<!ATTLIST m:Type
        Id ID #REQUIRED
        Definition CDATA #IMPLIED
>

<!-- Element object; descendant of list type -->
<!ELEMENT m:Element (m:Names)>
<!ATTLIST m:Element
        Id ID #REQUIRED
>

<!-- Unit object; measurement or currency -->
<!ELEMENT m:Unit
        (       m:Names
```

```
        ,       m:Description?
        )
>
<!ATTLIST m:Unit
        Id ID #REQUIRED
        Definition CDATA #IMPLIED
>

<!-- Index object; array dimension subscript -->
<!ELEMENT m:Index (m:Names)>
<!ATTLIST m:Index
        Id ID #REQUIRED
>

<!-- Names of object -->
<!ELEMENT m:Names (m:Name*)>

<!-- Name of object in given language and form -->
<!ELEMENT m:Name (#PCDATA)>
<!ATTLIST m:Name
        Locale CDATA #IMPLIED
        Plural CDATA "false"
        Abbrev CDATA "false"
        Dummy CDATA "false"
>

<!-- Description of object -->
<!ELEMENT m:Description (m:TextDocument*)>

<!--Text of description -->
<!ELEMENT m:TextDocument (#PCDATA)>
<!ATTLIST m:TextDocument
        xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink"
        xlink:href CDATA #REQUIRED
        xlink:type CDATA #FIXED "simple"
        xlink:show CDATA #FIXED "onRequest"
        xlink:actuate CDATA #FIXED "embed"
        Format NOTATION (xhtml) "xhtml"
        Locale CDATA #IMPLIED
>
```

Figure 59 – DTD dataflow model (significantly simplified)

A dataflow model document consists of different parts, with very different objects and purposes. The dataflowml framework defines dedicated DTD and XSD schemas for each part, and combines the parts together, as needed, when storing model documents. Below is an example, where the DTDs for the model and the view parts are combined into a complete document:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE Document [
  <!ELEMENT Document (m:Model, v:View)>
  <!ENTITY % ModelDtd SYSTEM "model.dtd">
  <!ENTITY % ViewDtd SYSTEM "view.dtd">
  %ModelDtd;
  %ViewDtd;
]>
<Document>
  <m:Model/>
  <v:View/>
</Document>
```

Figure 60 – DTD dataflow document, built from DTDs for its parts

5.4 Extending the framework

When used by Dynaplan, the framework XML is already extended by adding more elements (object types, and other elements) and by adding more attributes. The same can be done by others who want to base their file format in full or in part on the XML version of dataflowml.

The framework specifies the definition of objects using a single property. Some vendors may have problems if just a property is provided for the object's definition. In such cases, the vendor can add sub elements or extra attributes to their XML elements defining the objects in question. Another approach, which I'd prefer, is that the definition syntax is expanded to capture all aspects of an object's definition.

Another challenge might be that vendors want to put all parts of a model into one, big XML document. This can be achieved by putting all the parts together in one file, instead of as local files inside a multi-part binary file, which is the solution used by Dynaplan Smia. Parts that are not XML (pictures, HTML, etc.) can be stored as data URLs inside the XML.

Smia has a command for exporting a model to a directory, where each individual part becomes a separate file inside a folder structure. This approach can be used also by others.

6 Characteristics of core system dynamics

People are talking about *system dynamics models* and *system dynamics software*. What is it, really, that makes a model a system dynamics model or software?

A system dynamics model can be defined as any quantitative model that represents stock-and-flow processes and contains feedback. Such models can, in principle, be created in almost any modelling or programming environment – even the spreadsheet.

What makes a product into a system dynamics software, then, is that it has built-in support for the concepts of stock-and-flow, that it allows feedback, and that it can carry out the numerical integration processes necessary to calculate the behaviour of such models over time.

6.1 The inner core

Stock-and-flow concepts belong to the area of math which deals with integration, which is one level up from the basic mathematical concepts of algebra. Algebra is a necessary requirement for being able to create useful models, i.e., the software must support addition, subtraction, multiplication, and division).

The following table is my attempt to define the smallest possible feature set of a system dynamics software.

Concepts	Immediate variables, stock variables, numerical integration, literals, and mathematical expressions involving variables, literals, and the operators plus, minus, multiply, and divide. (The power operator might also be defined part of the smallest core).
Syntax	A grammar for expressing the above concepts as text and/or graphics. The selection of integration dimension and options for the integration algorithm must be available to the user to specify as part of the grammar and/or as part of the associated GUI of the application.
Semantics	The meaning of the grammar must be unambiguous and in harmony with the explicit and implicit requirements of the concepts mentioned above, so the value the variables in any model following the syntax can be calculated (by humans as well as by software).
Software	The system must implement automatic algorithms for reading and interpreting any model that satisfies the syntax, producing the correct results. If a model contains stocks, a time series for the stocks and all variables that depend directly or indirectly on stocks, must be created.

Figure 61 – Minimum requirements for being a system dynamics tool

Well known system dynamics packages, such as DYNAMO, Vensim, iThink, Constructor, and Powersim Studio are examples of software fulfilling the above requirements. In addition, these tools to a varying degree provide additional concepts and related functionality.

6.2 Adding the concept of choice

Conditional statements are useful extensions to the framework described above. Conceptually, conditional statements imply that a logical (Boolean) data type is introduced. However, this can be hidden from the explicit language syntax by putting the logic into functions operating on numbers. This was done by DYNAMO through its CLIP and SWITCH functions, which are defined like this (I believe, but this should be checked):

Expression	Meaning
CLIP(P,Q,R,S)	**if** R >= S **then** P **else** Q
FIFGE(P,Q,R,S)	"First If Greater than or Equal". (Meaning as above).
SWITCH(P,Q,R)	**if** P=0 **then** Q **else** R
FIFZE(P,Q,R)	"First if zero". (Meaning as above).

Figure 62 – Dynamo's introduction of conditions only involving numbers

Later systems have introduces relational operators (<, <=, =, <>, >=, >) and logical operators (and, or, not, xor), which return and/or operate on logical values. If a logical type and logical literals are not explicitly added to the syntax, the resulting language becomes in conflict with the concepts of logic.

It is interesting to see that DYNAMO, which has no logical data type, managed to manoeuvre around violation of concepts by smart[17] choices of functions. However, introducing the logical type and related operators is definitely the right decision for a modern dataflow language.

[17] The choice of functions is smart since it fulfils mathematical and logical concepts, but it is quite unintuitive for modellers to use.

6.3 Adding lookup functions

System dynamics is often used on phenomenon where the modeller believes there is a relationship between certain variables, but where the mathematical formulation of the relationship is unknown or quite difficult to formulate mathematically. In such cases it can be convenient to draw the graph describing the relationship rather than coming up with a formula that happens to produce a similar graph.

Lookup functions are also called *table functions* or *graphical functions*. These terms reflect how the functions can be visualized, either as a table of numbers or as a graph. In general, the name of an object should not reflect its role or its visualisation, but rather its nature (what it *is*). So maybe *lookup function* is a better term, even if that too is not perfect?

There are many variations of lookup functions, depending on how the fix points of the function's graph are specified, how points that fall between fix points are handled (interpolation), and how points outside the range of fix points are handled (extrapolation). In general, the name and the parameters of a lookup function needs to specify the following information:

Expression	Meaning
x values	The input domain of the function.
y values	One value for every x-value
interpolation	How to deal with input values that are between the given x values. Possible options include *none*, *linear*, *curve*, *look backward*, and *look forward*.
extrapolation	How to deal with input values that are before the first given x value or after the last one. Possible options include *none*, *linear*, *horizontal* (on both sides of the graph).

Figure 63 – Necessary implicit or explicit information for a lookup function

In a program such as Smia, the x and y values can be given together as an array with x as dimension and y as elements. Another approach is to specify the x values using a type and the y values as standard vector (array with one dimension, 1 to *n*). If types are not supported, the x dimension can be specified as another standard vector. If the x-values are equidistant, it is also possible to specify the values using only two inputs: x0 and dx, where x0 is the first x value and dx the distance between successive values. The number of values is equal to the length of the y array. Interpolation and extrapolation can be part of the function name, or specified as additional parameters.

For a language without array variables, it is still possible to implement lookup functions, either by introducing the concept of an array literal, or by supporting "greedy" functions that take a varying number of parameters. In the latter case, the y values can be listed last, occupying as many parameter positions as necessary. The array approach is the better solution, since it treats the vector as one entity and since it points forwards towards a full support for arrays. Below are some alternative ways to invoke a lookup function with the fix points (0,10), (10,15), (20,20), and (30,22):


```
xtable={0, 10, 20, 30}
ytable={10, 15, 20, 22}
xytable={0 to 30 step 10| 10, 15, 20, 22}
xdim=0 to 30 step 10
x = 5
x0 = 0
dx = 10
y1 = lookup(x, x0, dx, table)
y2 = lookup(x, x0, dx, {10, 15, 20, 22})
y3 = lookup(x, xtable, ytable)
y4 = lookup(x, {0, 10, 20, 30}, {0, 10, 20, 30})
y5 = lookup(x, {xdim|0, 10, 20, 30})
y6 = lookup(x, xytable)
y7 = xytable(~x)
```

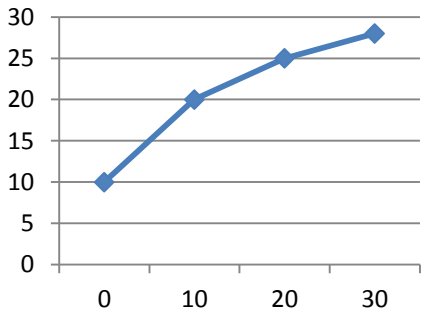

Figure 64 – Alternative ways to express a lookup function

The graph to the right displays the relationship between x (horizontal axis) and y (vertical axis) as described by the four fix-points, assuming linear interpolation. The seven equations for y all produce the result 12.5. Here are some comments to the different equations:

y1 – The x values are given as x0 and dx. The y values are input via the named array variable ytable.

y2 – Same as above, except that the y values are specified using a literal standard vector.

y3 – Here both x values and y values are given via named array variables.

y4 – Here both x values and y values are given as literal standard vectors.

y5 – In this case x and y values are given together as a vector with x values as its dimension and y values as elements.

y6 – Same as above, this time referring to the named xy table instead of using an array literal.

y7 – Smia's compact syntax for array lookup using interpolating subscripts. The expressions xytable(x) will return *empty* if x is not member of the dimension of xytable. By adding ~ in front, linear interpolation is used by the subscript. It is a compact representation, and a more general solution than using lookup functions, since the subscript alternative can be performed over multiple dimensions simultaneously. As an example, models can perform interpolation into sampled 2D or 3D space.

My suggestion for a core language definition for system dynamics is that the solutions for y1 and y3 become a minimum requirement. (The syntax can be different, but the semantics must be provided).

6.4 Adding frequently used functions

The core language should implement functions that are part of many existing system dynamics models, for example as presented in John Sterman's book: *Business Dynamics*. Here is a list of functions that I think will be necessary to include for the language to be useful for the community:

Category	Functions	
Logarithm, exponential, and square root functions	ln(x) log(x, base=10) log10(x) exp(x) sqrt(x)	
Trigonometric functions	sin(x) cos(x) tan(x)	
Statistical functions	min(a,b,...) max(a,b,...) min(array) max(array)	(* Greedy versions *) (* Array versions *)
Delay functions	delay(x, lag, order=1, leak=0, initial=x) delay material(x, lag, leak=0, initial=x) smooth(x, lag, order=1, initial=x) delay information(x, lag, initial=x)	(* Nth order material delay *) (* Infinite order material delay *) (* Nth order information delay *) (* Infinite order information delay *)
Test input functions	pulse(x, start, duration) pulse(x, start, duration, interval, stop=stoptime) ramp(slope, start, stop=stoptime) step(height, start)	
Random process functions	uniform process(min, max) normal process(mean, stdev, left=-infinity, right=infinity)	

Figure 65 – Commonly used functions

Note some functions have different names in other tools. Vendors can pick other names, of course, as long as the semantics is possible to reproduce.

Regarding the random functions, the chosen naming highlights that a new sample is drawn for every time step. The alternative is that a random sample is generated initially, and kept for the rest of the simulation. Dynaplan Smia provides two separate sets of random functions for this reason.

6.5 Further extensions to the core

This must be up to the individual vendors to decide. Obvious axes to expand along include the following (in no particular order):

- language support (locales)
- documentation of models and model objects
- units
- hierarchy
- arrays
- additional types
- additional functions (there is practically no limit; add based on customer needs)
- more analyses (optimisation, uncertainty, and more)
- object-oriented features (components)
- co-models (models that run in parallel, possible with separate time horizon, exchanging information)

Maybe some of the above points should be moved into the definition of the core? Maybe some of the suggestions do not fit for the customer segment(s) addressed by some vendors?

As long as there are multiple, independent, vendors I expect that the concepts and semantics – not to mention the syntax – used to implement extensions listed above, will vary quite a bit. This is not necessarily a bad thing, as it allows feature to emerge, develop, and maybe converge at a later stage, when they become mature.

Current versions SD software already contains features that I consider to be extensions compared to the core SD framework. Such extensions should be possible to implement within the framework. As an example, Vensim's support for multiple equations to define an array can be mapped into a single equation, following the example relating to spreadsheets in Figure 9 – Smia equation representing worksheet. Partial Vensim array equations would be collected inside one array definition, using syntax like this, for example:

Vensim

```
location : toledo, bismark ~~|
capacity utilisation[toledo] = production * fraction production to toledo / capacity[toledo] ~~|
capacity utilisation[bismark] = production * (1 - fraction production to toledo) / capacity[bismark]
~Dmnl
~The fraction of capacity used.
|
```

Dataflowml

```
dim = [toledo, bismark]
capacity utilisation =
{    toledo => production * fraction production to toledo / capacity[toledo]
|    bismark => production * (1 - fraction production to toledo) / capacity[bismark]
} as real
```

Figure 66 – Transforming multiple Vensim array equations into one

For the comparison, Smia would represent *capacity utilisation*[18] like this:

```
capacity utilisation = { location
|    production * 'fraction production to toledo' / capacity
,    production * (1 - 'fraction production to toledo') / capacity
} as real
```

6.6 Data dictionary for system dynamics

Going forward, it might be a good idea to consolidate the naming used by different vendors and authors when referring to concepts. This has probably been done before, but let me provide my own list of terms that relate directly to the topics of this paper.

accumulator – See stock

alias – Alternative name for an object. I prefer the concept of name form, with well-defined role, instead. (Example, singular, plural, abbreviation. See chapter 4.4.1)

array – 1) An un-named array that can be used in expression. Example: {1,2,3}. 2) A variable with a value that is an array.

[18] Space characters in a name placed to the left of the equal sign do not lead for the need of quotes in Smia.

auxiliary – A variable without inertial, i.e., the counterpart of a stock. Its value is computed by evaluating its definition in the current state of the simulation.
Boolean – See logical.
comodel – Self-contained model or model fragment, with its own integration settings, running in parallel with the main model, exchanging information with the main model at set intervals.
component – Named object with an interface and an implementation. Runs under the same integrator as the main model.
constant – A read-only variable with a definition that evaluates to a value that does not change over time (during the simulation).
converter – See auxiliary.
dimension – Value domain for the keys (or subscripts) of an array. A dimension is defined in terms of a type, and can in addition involve an index variable (like in math).
dimensional checking – See subscript checking.
double – See real.
element – 1) Member of an array, located by its subscripts (keys). 2) Named member of a list type.
enumeration – See list.
flow – 1) Part of a stock's definition, holding the amount of change that will take place to the state of the stock as the simulation advances. 2) A role a variable has if it is referred to from a flow expression. A variable is not a flow, but it can act as a flow. A stock can play the role as a flow of another variable. Example: Acceleration, speed, distance model in mechanics. It is conceptually wrong to introduce an object type called flow. (Pedadogy!)
group – An organizing concept, similar to a submodel, but with no support for nesting. A concept that should be phased out, in my opinion.
horizon – Type defining the simulation dimension. Can be expressed as a type, or as three variables (start time, stop time, and time step).
immediate variable – See auxiliary.
level – See stock
list – A type holding named elements. Can be hierarchical.
literal – A textual representation of a member of a type. Example: 123.
logical – Data type with values *false* and *true*, and sometimes also *indefinite* and *empty*.
macro – Question: Is this a template for parametric text substitution (as in C) or is it something more structured??
matrix – An array with two dimensions. A matrix with dimensions 1 through N by 1 through M is called a standard matrix.
measurement – See unit.
mixed – A data type that can hold values of any simple type, such as real, integer, logical, or text.
namespace – 1) An hierarchy for managing names, as opposed to objects. In my opinion it should not be used in the context of dataflow models. Use object hierarchy instead. 2) Some computer languages define separate symbol tables for objects of different type, for example **types** and **variables**. This is very awkward for end-users, as it introduces subtleties that can be hard to understand. It can also make model translation, and the semantics of paths more complex to define, implement, and understand.
range checking – Check that the value of a variable is within its type. Somewhat related to Vensim's reality check.
rate – 1) A value that represents a quantity over time. 2) A variable that acts as a (non-additive) flow is by definition a rate. A rate has the unit Q/time, where Q is any unit and time is the unit of the time horizon. Since Q can be any unit, it cannot be assumed

that a rate has a unit with (the unit of) time as a divisor.
real – Data type for real numbers.
scalar – 1) Value that is not an array. 2) Variable with a scalar value.
state variable – See stock.
stock – A variable with inertia, i.e., a state that is initialized at the beginning of the simulation and changed during the simulation via its flows.
string – See text.
submodel – An named object without definition, which only purpose is to build an hierarchy.
subscript – Parameter to an array lookup. Similar to a parameter of a function. Subscripts can be types or scalars. Example: vector(i).
subscript checking – Check that subscripts are within bounds when performing array lookup.
text – Data type holding character sequences (strings).
type – A (possible infinite) set of values. The set of real numbers is an absolutely necessary type for dataflow models. Logical operators and conditions require the logical type. It is conceptually wrong to write an expression like a<b+1. The semantics of the grammar should deny such non-sense. Types are used to define dimensions of arrays, the value type of a variable, and the integration dimension of a simulation.
unit – Measurement, or meaning, of a number. Example: item, second, miles per hour, kg. A unit can be a point unit or a normal unit. Point units have an origin other than zero.
unit checking - The term *dimensional checking* is sometimes used for validation of measurement units. I suggest that *unit checking* is used instead, in order to avoid confusion. Dimensional checking could also mean check that all subscripts are within range when accessing arrays. For this activity it is more precise to use the term subscript checking.
variable – A named object with a value that is determined by evaluating its definition, which is an expression. A variable can be a stock or an auxiliary.
variant – See mixed.
vector – One-dimensional array. A vector with dimension 1 through N is called a standard vector.

Figure 67 – A tiny data dictionary

