

LIBPF: A LIBRARY FOR PROCESS FLOWSHEETING IN C++

Paolo Greppi
Department of Environmental Engineering
University of Genoa
Via all'Opera Pia, 15/a 16145 Genoa (GE), Italy
E-mail: paolo.greppi@diam.unige.it

KEYWORDS

Process flowsheeting, modelling, continuous chemical processes, object oriented

ABSTRACT

Process flowsheeting is an activity nowadays routinely performed using dedicated commercial programs, which implement the required computations and provide the process engineer with a user-friendly graphical interface.

We discuss the different approaches for process flowsheeting, and present our own approach, LIBPF, based on describing the model directly in the syntax of a high-level object-oriented programming language.

The scope of the proposed approach is limited to steady-state or dynamic modelling of first principle, process-wide concentrated parameters models for continuous chemico-physical processes, with a special focus on on-line applications.

INTRODUCTION

Process flowsheeting (Westerberg et al. 1979) is an approach to concentrated parameters modelling based on representing the process as a directed graph. It has the capability to represent virtually any process, but historically it is an approach mostly used with this name in Chemical Engineering; other domains where this paradigm is currently applied with different names are stock-flow diagrams in econometric models, “network structure” in LCA (Life Cycle Assessment), Heating Ventilation and Air Conditioning (HVAC) modelling.

In practice process flowsheeting can be used to develop off-line models (for example in the design phase), or to develop models as part of solutions for on-line diagnostic, data reconciliation, soft sensors or advanced control.

This contribution focuses on on-line applications; in the following we will define the basic requirements for this kind of solutions, discuss alternative frameworks for implementing them, and finally illustrate the design of our own approach.

REQUIREMENTS FOR SOLUTIONS BASED ON MODELLING

Information Technology (IT) solutions based on modelling and designed to run on-line within an industrial environment have to fulfil a minimum set of quite general requirements:

1. Be tailored on customer processes
2. Integrate in customers' IT systems
3. Be reliable
4. Be maintainable

Requirement 1: Customization

Each project will be different; and each customer will require a solution perfectly fit for their own problem. For this reason the solution provider needs a flexible and effective development tool to encode the model.

Requirement 2: IT integration

As (Brooks, 1995) points out, much of the complexity in a “programming system” arises from the need to interact with other system components.

The **hardware system** components range from the embedded microcontroller to the Programmable Logical Controller (PLC), to industrial PCs and to Distributed Control Systems (DCS).

The **software system** encompasses enterprise databases, real-time databases, SCADA (Supervisory Control and Data Acquisition), web browsers and office productivity suites.

There is a number of standardized, cross-vendor interfaces to tackle this diversity, such as OLE for Process Control - OPC (OPC Foundation, 1999) and Open Database Connectivity – ODBC (Microsoft Corporation, 2004).

Requirement 3: Reliability

Reliability is critical for software applications which might impact industrial processes with the associated Health and Safety implications. Translated in terms of actual software requirements, the program should:

- validate inputs
- provide correct results if a solution exists

- log errors (communication, data consistency, computation)
- never crash
- not leak any memory (an important requirement for long execution times).

Requirement 4: Maintainability

The time scale for the life of an installation in the process industry is measured in decades, in some cases even half century. For IT, this is way beyond the normal time horizon: to upgrade, update and recompile a program after 25 years is a challenging task.

We believe the only solution is to allow the customer to own or have access to the source code of the application, and to own or freely access the development tools.

TOOL AND LANGUAGES OPTIONS

There are a number of more or less dedicated, more or less general purpose tools and programs one can use to create modelling solutions for industrial applications.

We can group them as follows:

1. Specific object-oriented tools for modelling: Modelica, gProms, ProSimPlus, ACM, ChemaSim
2. Mathematical toolboxes: Matlab, Maple, Mathematica, Maxima, Derive
3. High level programming languages: FORTRAN, C++

Specific object-oriented tools

These tools are very specific and developed explicitly for applications in Chemical Engineering.

Their **advantages** are:

- High efficiency in calculations
- Short development time for the solution, since generally libraries of predefined models are available and likely to be well tested.

Their **drawbacks** are:

- The solution provider and the customer of the solution by choosing one of these tools bind themselves to the software vendor in an unsymmetrical relationship
- It is relevant for academia that you can do no real research with these tools; in fact their development process is always targeted towards specific present industry needs. To make an example, by the time fuel cells are no more a research topic you'll get full support for fuel cells in these tools.
- Being special purpose tools they have a small user community; and this in turn is very bad news for the reliability, since it increases the risk for the average user to encounter serious bugs.

Mathematical toolboxes

These tools are general purpose packages for symbolical or numerical computations.

Their **advantages** are:

- Reliable and easy to use, thanks to a user community a couple of orders of magnitude larger than specialized tools
- Interpret proprietary languages, but they are so widely used that these languages are less likely to go out of control.
- They can export code via C converters so that there is no performance penalty to pay when the model is ready for prime time.
- Overall the dependence on the tool provider is less tight.

Their **drawbacks** are:

- The libraries of models do not typically cover unit operations required in the applications we are interested in
- No objects and data structures specific for process engineering; for example they have no understanding of the very concept of process flowsheeting.

High level programming languages

Writing from scratch a program in a high level general purpose programming language to solve a practical process flowsheeting problem is a bold design choice. This task was not realistic with a language of the generation of FORTRAN77, targeted to the manipulation of integers, real numbers, characters and files. With the appearance of more advanced programming languages, which emphasize object orientedness and code reuse it is now a realistic and possible alternative.

The **advantages** of this approach are:

- Language is vendor-independent and with a huge and varied user community; this guarantees portability and reliability
- Computation can be faster if a compiled language is used – actually there is a compromise between fast and maintainable code which should be traded off.

The **drawbacks** are:

- Special competencies are required
- There is a steep learning curve required to be able to encode a model
- Maintainability is a potential risk, in case the complexity runs out of control.

THE BUSINESS CASE FOR C++

We argue that the best approach to develop process flowsheeting models as part of solutions for on-line industrial applications is to encode them using a high level programming language, and precisely C++.

The start for the paradigm for Object Oriented Programming (OOP) dates back to (Dahl, Nygaard 1966), and it is not by chance that its story begins with modelling.

It took 36 years to standardize an industrial-strength programming language (ISO 1998) and it took another 7 years for compilers to catch up and implement the complete standard (Microsoft 2005) (GNU 2005).

We believe as of today C++ is the right language to reinvent the wheel (Buzzi Ferraris 2000) in the domain of process flowsheeting because:

1. It is based on an international standard
2. Industrial-strength compilers are available
3. It can manipulate complex objects
4. There is a wealth of available third-party libraries available for reuse
5. There is much ongoing research on migrating numerical libraries from FORTRAN to C++
6. We see the first applications to real-time systems, migrating from C to C++

Of course the C++ programming language is not the “silver bullet” (Brooks 1995), it is just an effective but imperfect tool for the task of process flowsheeting.

We note that the possibility of using C++ for flowsheeting is related to the idea of using C++ to implement a Computer Algebra System, as in (Bauer et al. 2002) and in (Kiat-Shi et al. 2000).

DESIGN OF LIBPF, A C++ PROCESS FLOWSHEETING LIBRARY

Component reuse

The LIBPF library has been written using available third-party software components, libraries and tools to the largest extent possible; this reduces the code size, increases reliability and decreases risks.

The external libraries and tools used are:

- STL (Standard Template Library) for containers
- Tapeless ADOL-C for Automatic Differentiation (for 1st order derivatives)
- boost::graph 1.33.0 for graphs
- Microsoft Data Access SDK 2.8 (ODBC)
- GMM++ 1.7 (Generic Template Matrix C++ Library)
- SUNDIALS (IDA 2.3.0) for DAE integrators
- graphviz for graph display
- Existing java / XML component for tree view in HTML

Objects required for flowsheeting

Process flowsheeting in C++ means a flowsheet is represented in memory as a parameterized graph, the edges representing the material streams connecting the vertexes and the vertexes representing the unit operations. Since any equation set can be interpreted as a parameterized graph, where the edges are the unknowns connecting the vertexes which in turn are the functions transforming the variables, we can view a

flowsheet as a particular type of object-oriented equation set, where the unknowns are not real numbers but rather vectors of real numbers with certain constraints (a material stream).

This approach means that we can use graph algorithms to analyze the connectivity and find the solution path.

The objects that can be parameterized on the **graph vertexes** should be arranged in a hierarchy according to their connectivity capability; we use class names such as `one_one` or `many_one` to represent the capability of having one inlet and one outlet, or one to infinity inlets and one outlet attached, respectively.

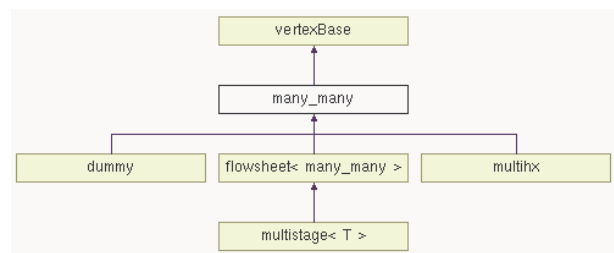


Figure 1: Subset of Class Hierarchy for Graph Vertexes

Figure 1 shows the class arrangement for `many_many`, the semi-abstract class representing the capability of having one to infinity inlets and one to infinity outlets attached; `multihx` which is the generic multistream heat exchanger and `multistage<T>` which is the generic co-flow / counter-flow multistage unit assembled from stages of type `T` are both derived from `many_many`. By deriving the class `flowsheet` itself from `vertexBase` we allow for recursion, i.e. `flowsheet-in-flowsheet`.

The objects that can be parameterized on the **graph edges** should build a hierarchy of stream types. The base semi-abstract `stream` class provides basic functionality for attaching and detaching, and dictates interfaces for flash calculations, property calculations etc. that concrete derived classes should implement.

Flexibility

The ways the different objects may be combined are infinite; the requirement for flexibility is best illustrated with a use case; suppose the user writes a model for an equilibrium stages reactive distillation column. The model has many options:

- how many stages
- which reactions are considered and where
- whether reactions are at equilibrium or with fixed conversion
- which equation of state is used to calculate the phase equilibrium and where

We certainly do not want to declare and explicitly define a class for each combination of the options. Templates could help, so if `genflash22` is the flash with two inlets and two outlets

`multistage<genflash22, 10>` could be a column with 10 stages.

To declare a class for each combination of the options, with or without templates, amounts to implement the design pattern **flexibility via inheritance**. The main advantage here is that the relationship between the two classes (base and derived) is a compile-time relationship (type safety)

The disadvantages are:

1. The type is not dynamical at run-time (lack of flexibility)
2. The number of classes explodes if we have to declare all combinations of number of stages, reactions, property settings

The alternative design pattern is **flexibility via delegation**, which uses a run-time relationship between two objects.

Going back to the reactive distillation use case, the column would be an instance of a `multistage<genflash22>` object; the number of stages is not anymore a template parameter. Instead each `multistage` instance contains a dynamical vector of pointers to `genflash22` objects, and the number of stages can be defined at run-time by resizing that vector.

Whether each stage is reactive or not, and which reactions are involved should also be defined at run-time. This is possible if `genflash22` objects contain a dynamical list of pointers to an abstract class `reaction`, which can be populated with dynamically allocated objects of the right type at run-time.

Lastly, the flexibility to set which equation of state is used to calculate the phase equilibrium on a stage wise level can be implemented if the connections between stages are of a template stream type so that `streamVL<ideal>` or `streamVL<SRK>` are possible constructs.

We note that with this arrangement the type is dynamical at run-time (Baumeister 2000, 2.3.2), so the class it instantiates does not uniquely determine the type and the extension of a specific instance.

Persistency

Once you have performed a conceptual analysis of the object hierarchy required for handling flowsheets and the related flexibility, the next step (even before the step of performing computations) is to device means of saving and restoring problems. The requirement to be able to restore the status of a previous run from disk to memory translates in the requirement that objects should be persistent.

Persistency options known in the field of Object Oriented Programming are:

- Serialization (to text, XML or binary file) as in Java or `boost::serialize`
- Serialization provided by an external register and broker: CORBA or COM or .NET
- Write specific persistency mechanism
 - store in a file
 - store in an object-oriented database.
 - store in a relational database.

We believe the database option is preferable, and based on the lack of a reliable off-the-shelf industrial-strength object-oriented database, the chosen design was to write from scratch a persistency mechanism which relies on an external relational database.

The variables are stored in the database in three flat tables, arranged by variable types (this amounts to a sort of serialization):

- Integer variables in ITBL
- Real variables in QTBL, and
- String variables in STBL

There is a master table (CATALOG) which contains the list of all objects, and the ID of this table has a one-to-many relationship to the CATALOGID field of each of the {I,Q,S}TBL, as illustrated in figure 2.

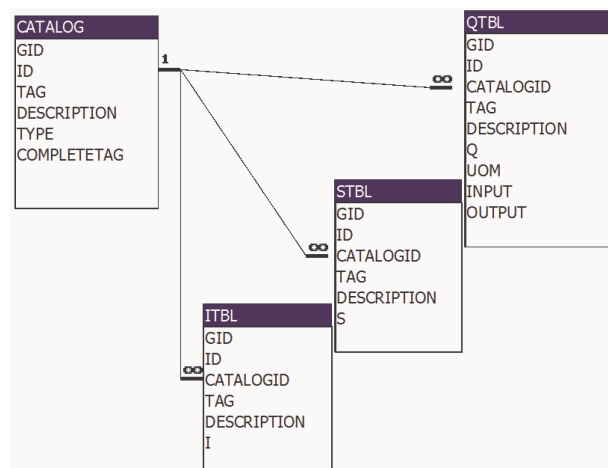


Figure 2: Data Structure Diagram

The recursion can be implemented using some of the integer variables of the container object to store the CATALOGID of the contained objects.

As we turn to the consequences of persistency to the design of the library, we see that object construction and access must be possible both with full compile-time type information as well as with run-time only type information. To implement this you need:

- An object factory
- Parameterize all construction options by simple persistent objects (integers or strings)
- Reflection

Reflection

Type reflection is the ability to programmatically inspect and use types, and it is required for:

- dynamic type manipulation
- persistency

Going back to the reactive distillation use case, suppose the user has defined a reactive first stage, with three reactions, the last one being an equilibrium reaction.

Setting the equilibrium constant for the equilibrium reaction would require a syntax like (recall that in C++ arrays are numbered from 0):

```
C1->stage[0]->reactions[2]->K=2.0;
```

Unfortunately this is not supported in C++ for classes based on flexibility via delegation, since the dynamical list of pointers `reactions` will contain pointers to an abstract class `reaction` which is not aware of any variable `K`, this variable being specific to `equilibriumReaction` (a class derived from `reaction`).

One work around would be the use of dynamic casts, but that makes the code unreadable. While waiting for support from the new standard and `<type_traits>`, a formalism to inspect a class instance for available members has been implemented, such that the assignment becomes:

```
C1->stage[0]->reactions[2]->Q("K")=2.0;
```

Failure to find a variable named `K` in `reactions[2]` would result in a run-time error as in case a dynamic cast had been attempted.

Derivatives

Derivatives are required for the resolution of non-linear equation sets (up to 1st order derivatives) and for optimization (up to 2nd order); within the scope of this work we only need 1st order derivatives.

Derivatives can be computed via numerical perturbation or analytically. Analytical derivatives can be hand-coded (or equivalently based on code generated with a computer algebra system), or obtained with automatic differentiation. Automatic differentiation can be performed with code transformation (i.e. using an additional compilation step), or through Operator Overloading.

The selected approach is Automatic Differentiation via Operator Overloading. The implementation is based on tapeless ADOL-C (Kowarz and Walther 2004), with a dense representation of the derivatives; in addition it provides means to switch context between different sets of unknowns.

We have found extremely handy that the automatic differentiation is performed by the compiler in a self-

contained way; this results in a streamlined code-compile-debug cycle during development.

Obviously this comes at a cost, since it is known that derivatives through operator overloading can be anything from a factor two up to an order of magnitude slower than derivatives obtained with code transformation.

The price is worth paying in the framework of a technical compromise between fast execution and a certain flexibility of the development tool, similar to interpreted languages and mathematical toolboxes.

Furthermore, as the code base matures, it will be possible to move the most performance-critical sections to hand-coded derivatives.

Status

The LIBPF C++ process flowsheeting library as of today is in active development; the current release is 0.5, consisting in 30,000 Lines Of Code (LOC).

The **thermodynamic** computations cover:

- ideal gas phase
- ideal vapour-liquid equilibriums (dilute systems)
- vapour-liquid equilibriums with cubic equations of state
- reactive mixtures: equilibrium reactions in the ideal gas phase

The library includes **unit operation models** for:

- mixer, 2 or more inlets
- flow splitter (tee), 2 or more outlets
- spawn (duplicates the inlet)
- fixed-yield separator, 2 or three outlet streams
- vapour-liquid flash
- isentropic compressor/expander
- reactive multi-stream heat exchanger, concentrated parameter
- concentrated parameter fuel cell with 2 or more streams, each supporting multiple equilibrium or fixed conversion reactions; one or more electrochemical reactions supported
- countercurrent non-reactive adiabatic HTU/NTU absorber/stripper
- flowsheet-in-flowsheet
- generic multistage co-/counter-current unit

The **solution mode** is sequential, with simultaneous convergence of recycles and of flowsheet-level process specifications (feed-back) using straight direct substitution or Wegstein-accelerated direct substitution.

The library is actively applied on **real-world cases**, and it was tested on the following applications, including absorption/stripping, low pressure gas cleaning / processing, biotech processes and fuel cell system modelling.

DEVELOPMENT AND QUALITY CONTROL TOOLS

To satisfy the requirement of reliability, there is a set of well-known generic techniques of software engineering specific for quality control, such as version control, literate programming and profiling.

To identify the source of problems quickly we have implemented a finely-tuneable diagnostic system, with verbosity that can be set at a global level, subsystem level (properties, unit operation, components and persistency subsystems) and function level.

As to the reliability requirements specific to engineering software, we have found that dimensional check of equations and consistency tests are needed.

Dimensional check of equations can be implemented as a compile-time check based on template metaprogramming like in (Brown 2001), but we have found that the very slow compile and the proliferation of types make its application unfeasible today.

We have therefore implemented a very simple run-time dimensional check, without support for rational exponents; the price to pay is that it slows down the execution but the checks can be turned off for the production executable.

Consistency tests like those based on the Gibbs-Duhem equation, *a posteriori* mass atom and energy balances, monotonicity tests, check of analytical derivatives against numerical derivatives etc. should be included in the test suite and are the best guard against conceptual mistakes.

CONCLUSION AND FUTURE WORK

It is feasible to encode process flowsheeting calculations in C++ using the LIBPF library; with this lightweight tool a substantial subset of the real world potential applications of concentrated parameters modelling to on-line industrial processes can be tackled. Thanks to the direct control of the source code of the library, and using widely available tools such as compilers and relational databases, tailored and reliable solutions can be developed, customized and maintained.

Planned developments to the proposed chemical engineering library include:

- Interfacing hand-coded derivatives with automatic differentiation via operator overloading, at least for frequently called functions like property calculations
- Sparse representation of derivatives
- Simultaneous solution of equations
- Three-phases flashes (VLLLE)
- Reactive flash with liquid phase reactions (electrolytes, oligomerizations and polymerizations)
- Solid phases

A compiled version of the library with headers and samples can be obtained with an academic license by mailing the author. The enhanced implementation of tapeless ADOL-C is available on request in source code form.

REFERENCES

- Bauer C., Frink A., Kreckel R. 2002. "Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language" *J. Symbolic Computation* (2002) 33, 1–12
- Baumeister M. 2000, "Ein Objektmodell zur Repräsentation und Wiederverwendung verfahrenstechnischer Prozessmodelle", Thesis, *RWTH Aachen*
- Brooks, F.P. 1995 "The Mythical Man-Month" *Addison Wesley Professional*
- Brown, W.E. 2001 "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation" In: *Second Workshop on C++ Template Programming*.
- Buzzi Ferraris G. 2000 "Prospects in the Field of numerical Analysis" Plenary Lecture, *ESCAPE-10*, May 7-10, 2000 Florence, Italy
- Dahl O.J., and Nygaard K. 1966 "Simula - an Algol-based simulation language" *Communications of the ACM*, 9(9):671 - 678
- GNU 2005. *GNU Compiler Collection 4.0*
- ISO 1998. "Programming Language C++" ISO/IEC 14882:1998
- Kiat-Shi T., Steeb W.H. and Hardy Y. 2000 *SymbolicC++*, *An Introduction to Computer Algebra Using Object-Oriented Programming* Springer-Verlag, London
- Kowarz A., Walther A. 2004, "Tapeless Forward Differentiation in ADOL-C" *Workshop on Automatic Differentiation*, Joint University of Hertfordshire/Cranfield University
- Microsoft Corporation, 2004 *ODBC Programmer's Reference*
- Microsoft Corporation, 2005 *Visual C++ 8.0*
- OPC Foundation, 1999 *OLE for Process Control / Data Access Automation Interface Standard* Version 2.02
- Westerberg A.W.; Hutchinson, H.P.; Motard, R.L. and Winter, P., 1979 *Process flowsheeting* Cambridge University Press, Cambridge.

AUTHOR BIOGRAPHY



PAOLO GREPPI was born in Vercelli, Italy and obtained his degree in Chemical Engineering back in 1995 at Politecnico di Torino. Currently he is freelance industry consultant and Research Associate with DIAM at Università di Genova.

He worked first at close contact with industrial plants as process engineer before joining a leading software provider for the industry where he was supporting modelling software. His position before trying the freelance approach was Engineering Director of a small-sized engineering company in Italy.

His group Web-page can be found at:

<http://www.diam.unige.it/pert/>