Port-based systems engineering of block models for control and simulation of Neutron Beam Instruments of the OPAL research reactor using UML/SysML

D. R. C. Kelly
The Bragg Institute,
Australian Nuclear Science and Technology Organisation (ANSTO)
PMB 1, Menai, NSW 2234, Australia
e-mail: darren.kelly@ansto.gov.au

Abstract The OPAL research reactor serves as a neutron source for diverse Neutron Beam Instruments (NBIs) undergoing commissioning throughout 2007 by The Bragg Institute of the Australian Nuclear Science and Technology Organisation (ANSTO) [1]. Port-based Systems Engineering (PBSE) using UML2.1.1[2] - and more recently SysML [3] with explicit physical flowport support - has been employed to support Model-Based Systems Engineering (MBSE) of NBIs, including reverse- and forward-engineering of Java components, and development of XML schemas for NBI hardware components and sensor-actuator device networks. Electronic systems, neutron flows, and vacuum systems are modelled as UML2 ports with custom notation and SysML flowports. The NBI models thus engineered are incorporated into a ModelServer system (implemented in Java+XML), which acts as a distributed, multi-client, control system façade that integrates low-level, logical device interactions (via a real or simulated control system) into high-level, physical instrument block models that support recursive control [4] and derived, controllable, physical system values.

Keywords: UML – SysML – systems-engineering – ports – neutron beam instrument

1 Introduction

Low-level, device-centric modelling and control of scientific instruments - while useful in its own right - supports only basic instrument operation modes, and does not easily support advanced modelling of physical aspects of an instrument such as geometry, materials, relative motions, physical flows, or beam physics, and it does not enable developers to take advantage of the power of graphical systems engineering of physical component models of instruments.

In order to support physical simulation, animation, high-level subsystem logic, and recursive control [4] of derived physical values and complex experimental modes, one does well to integrate a device-level electrical/logical view into a hierarchical, physical, component-based systems model with subordinate logical channels. I describe here how I have employed *Model-Based Systems Engineering (MBSE)* using UML2.1.1 [2] (and more recently SysML1.0 [3]) to support such modelling and control of *Neutron Beam Instruments (NBIs)* of the OPAL research reactor at ANSTO [1], in particular exploiting control-, data-, and flow-ports to drive the modelling and development.

While the examples are taken from NBIs, the notations and modelling principles presented here are immediately applicable to a wide variety of scientific instruments and other systems engineering applications

Instances of the NBI models are served by The Bragg NBI ModelServer system, which can run in standalone simulation mode with dummy controllers, or via a networked control system to monitor and control NBIs live. The ModelServer is Java5-based [5], so UML modelling and notation conventions are tuned for Java.

The focus of this paper is on presenting practical ways to organise and present structural and diagramming aspects of port-based systems engineering models using UML, drawing on real-world experience modelling and controlling NBIs at OPAL[1]. Particulars of controlled devices, control system injection, recursive control, geometry, simulation, and animation will be the subject of future papers.

All modelling and UML engineering work presented here was performed with the Magicdraw UML[1] tool (versions 11.0 through 12.5EAPbeta2) and its developmental SysML plugin (versions 1.0 through 1.1EAPbeta2).

2 Port-based Systems Engineering (PBSE)

I employ the term *Port-Based Systems Engineering (PBSE)* to refer here to a type of MBSE where a physical, electrical, or logical interpretation of ports is employed to drive and organise systems modelling and software engineering processes. This strategy lends itself immediately to the modelling of NBIs, where at least the following port categories can be identified:

1. *Electrical and digital ports (control ports and data ports)*: will be handled here as UML2 ports.
2. *Physical access ports*: such as feedthroughs for cables of electrical systems.
3. *Vacuum system flowports*: some neutron conditioning elements are enclosed by vacuum systems to reduce scattering of neutrons out of the beam.
4. *Neutron beam flowports:* the neutron beam can be modelled as flowing from a source (which in the case of OPAL is a research reactor [1]), through neutron conditioning elements (such as collimation, monochromation, pulsing, focussing), then through an interaction region (where the beam undergoes diffraction and/or reflection[2] and/or transmission), and into a neutron detector and/or beam stop.
5. *Environment flowports:* for example, specific gases are introduced into the a volume containing the sample being studied to establish a controlled sample environment.
6. *Radiation ports:* an interpretation of physical gaps in an NBI permitting radiation to exit the instrument, of interest to radiation surveys.
7. *Light ports:* porthole windows to enable examination of elements contained in vacuum regions.

It will be shown that PBSE supports different viewpoints of an instrument, of which the neutron beam flowport viewpoint is the primary viewpoint for NBIs. For example, a beam-centric representation of the monochromation systems of the diffractometers at OPAL (Echidna, Wombat, and Kowari [1]) will be shown

1 http//:www.magicdraw.com

2 Reflection here applies to reflectometry at low angles of incidence; back-scattering from a studied sample or neutron conditioning elements is not considered, although it may be of interest to – for example – radiation surveys.

to be differently ordered than one that emphasises the order of assembly of its components.

3 Notation and conventions

A significant aspect of the work presented here is a detailed recipe for organising systems engineering models and simulations, which recipe is supported by notations, conventions, and diagramming recipes (some of which are tuned for use with Magicdraw UML). Readers are encouraged to take particular note of the block <<wrapper>> Component strategy and the use of navigation points to promote fluent systems diagramming, to the immediate benefit of a wide range of UML-based systems engineering modelling tasks. Attention to the introduction here of these notations and recipes will be rewarded by a better understanding of the examples of applications to the OPAL NBIs [1]that follow.

3.1 Metaclasses and Stereotypes

Throughout this document nouns used as metaclasses of UML2 and SysML may be capitalised, including when used as plurals. Stereotypes used as nouns may be embedded using <<..>> notation in sentences. Thus:

> "The type of a Port used as a <<flowport>> .."
>
> ".. blocks may be logically wrapped by Components .."

A *UML2 Class* used as a systems block shall be stereotyped as a <<block>> (since the author uses a custom, SysML-like profile for UML2), whereas a *SysML Block* will be capitalised as though it were a true metaclass. Similarly, *SysML Flow* and *Flowport* shall be capitalised like true metaclasses. The distinction is important here, because UML2 was adapted by the author for systems engineering using a custom profile with SysML-like stereotypes long before the Magicdraw UML SysML1.0 profile was used.

The words *class* and *interface* are written lower-case when referring to Java source, whereas *UML Class* and *UML Interface* are UML elements (which correspond to Java under reverse- and forward-engineering).

3.2 Flowport notation for systems engineering in UML2

Already with the UML2 Port one can achieve a degree of PBSE, however both the lack of explicit support for indication of flow direction, and the notational need to distinguish software ports from systems ports, present a significant challenge, so a notation was developed to assist these distinctions, which notation was tuned to also work well with Java5. Most of the work presented here precedes the author's awareness of the the SysML effort [3].

The following notation for flowports was found to read well in UML2 diagrams and to map well to a Java implementation, although it does break with Java coding convention in some respects, especially concerning use of lower-case class and interface names, used here to indicate <<flow>> types and <<flowport>> types.

Flows are formally stereotyped as <<flow>> types and are written in lower case thus:

- **air:** atmospheric air flow (a special case of a gas flow)

- **n:** abbreviation of 'neutron beam flow' for brevity in the NBI context

The base port notation convention (without a specific <<flow>> type) is (see also Figure 2):

- **p$ / p$_**: abstract flowport interface and default abstract implementor class
- **i$ / i$_**: <<in>> flowport interface and default implementor class
- **o$/ o$_**: <<out>> flowport interface and default implementor class
- **io$ / io$_**: <<in-out>> flowport interface and default implementor class

The rationale for the notation is as follows:

1. UML Classes are used for <<flow>> types rather than, for example, UML DataTypes - to assist correspondence with reverse-engineered port classes implemented in Java in the Bragg systems engineering framework.
2. The lower-case first letters (`i,o,io,air,n`) help distinguish the <<flow>>s and <<flowport>>s from other engineering elements (software engineering classes and systems enginering blocks).
3. The '`$`' after the <<flowport>> type announces that it is a <<flowport>>, with the <<flow>> type to follow (see below for examples with named <<flow>> types)[3].
4. The trailing underscore '_' means here *default implementor class*, a convention used throughout the author's software engineering frameworks. It combines well with the common Java practice of using a trailing underscore for private variables, thus:

```
private Thing_ thing_; //trailing underscore and explicit type

public Thing getThing() {
      if (thing_== null) {
            thing_ = new Thing_(); //explicit reference to class[4]
            // configure the thing
      }
      return thing_;
 }
```

   It is also far more concise than other popular notations such as `Thing/ThingImpl` (which quickly makes UML diagrams hard to read, especially when used for port types), or `IThing/Thing` (which can confuse scientists not familiar with UML, and the `'I'` may be confused with *in/input*).

Although of interest to the targeted Bragg software simulation and control framework in Java, the UML Interfaces of flowports do not play a significant role in the aspects of systems engineering of NBIs presented here, so in what follows usually only the flowport implementor Classes will be shown, and they will simply be called *flowport types*, with the understanding that an implementor Class is always used as a <<flowport>>[5].

3 It has not been observed that the `'$'` used for flowports clashes with its use for inner classes in Java when compiled.

4 Of course in many cases one does well to instead obtain the implementation from an injected factory.

5 While interface types for ports are permitted they are not used in this work.

Ports may be notationally parametrised. Given a <<flow>> quantity of template parameter flow class 't' the parametrised flowport interfaces and classes are:

- **`i$<t>_`**: <<in>> flowport class
- **`o$<t>_`**: <<out>> flowport class
- **`io$<t>_`**: <<in-out>> flowport class

So for neutron beam flows (abbreviated as <<flow>> `'n'`) we have (see also Figure 3):

- **`i$n_`**: <<in>> neutron beam flowport class
- **`o$n_`**: <<out>> neutron beam flowport class
- **`io$n_`**: <<in-out>> neutron beam flowport class

The base flowports types may be parametrised (”templated”) using Java5 generics [5], and the Bragg NBI ModelServer framework offers generics support throughout. However, the base implementors cannot model specific physical flow properties (such as the velocity distribution of a neutron beam, or pulse characteristics); for that explicit <<flowport>> types for specific <<flow>> types are required[6].

In many cases <<flowport>>s may remain anonymous (or the names need not be shown) as the above notation reads well when no name is offered; in other cases the role may be indicated explicitly to suggest the transformation of neutron beam flow caused by a block: an out <<flowport>> from an attenuator might be named `'oAttenuated'`. Since neutrons are the primary <<flow>> of interest for NBI modelling, the type of the neutron <<flowport>>s is often omitted in diagrams, making them clearer for NBI scientists.

3.3 Logical and graphical support for systems engineering and software engineering using block <<wrapper>> and <<part wrapper>> Components

One must take care to distinguish “systems engineering” (using UML2 Classes stereotyped as <<block>>s or SysML1.0 Blocks[7]) from “software engineering” (as used in the Bragg ModelServer for simulation and control). Especially UML Classes reverse-engineered from Java classes need not correspond exactly with the SysML Blocks they represent/simulate.

1. Example: a SysML Block may have value properties representing physical quantities, each with a rich value type with a physical unit and dimension, and SysML provides for notations showing the default values and units of such rich value properties specific to a part property typed by a Block. In Java, this may be handled by a rich Value class, however the default for each physical value field must be handled otherwise, such as via a class-level (static) constant.
2. Example: a SysML Block explicitly supports SysML Flowports with direction and provides notation for that purpose. Related Java classes may (as illustrated above) use other notational devices to

6 It was also found that the reverse-engineered, parametrised generic port notation in Magicdraw UML did not appeal to most neutron beam scientists not familiar with UML, whereas an explicit port notation was more readily understood.

7 SysML implemented in a particular tool may also be modelled using a profile with a <<block>> stereotype.

handle directional flowports, as in the `flowport` package from the Bragg systems engineering Java framework.

3. Example: a Java interface/implementor pair corresponding to a SysML Block may have additional attributes and operations required for participation in a software framework, which features are unknown to the SysML Block in the UML/SysML model.

Such situations can be addressed by *logically wrapping* UML classes and SysML blocks with a UML2 Component for each abstracted[8] real-world system block being modelled, simulated, controlled, and animated. Such a *block <<wrapper>> Component* also provides for powerful logical views and graphical organisation of diagrams, without affecting the underlying physical packaging of the systems engineering blocks and/or software engineering Classes.

Two cases will be considered here:

1. Case: UML2 adapted for systems engineering:

   - 1 *(abstracted) real-world system block* with Java-friendly name.
   - 1 *block <<wrapper>> Component* (UML2) named after the real-world block.
   - 1 *reverse-engineered Class* (UML2) - and its matching Java class file - that realizes its paired interface. This will be called here the *default block <<implementor>>* Class. It is named after the block with a trailing underscore '_'.
   - 1 *reverse-engineered Interface* (UML2) - and its matching Java interface file - for its paired <<implementor>>. This will be called here the *block interface*. It is named after the block.

   In this case the <<implementor>> Class may be also stereotyped by a custom UML2 <<block>> stereotype, and the distinction between "systems engineering" and "software engineering" is blurred (yet mitigated by the use of the block <<wrapper>> Component for carrying systems engineering data).

2. Case: as above, and with additional SysML Block. Many SysML tools offer a <<block>> stereotype in a SysML profile, so in this case the <<implementor>> Class is not stereotyped as an <<<block>>, to distinguish "systems engineering" on the SysML Block from "software engineering".

In both cases the Java *implementor-and-interface pair* may have been forward-engineered from UML (if only as stubs) or they may have been entirely hand-coded; they are at least reverse-engineered "back into their block wrapper". The block <<wrapper>> Component recipe is illustrated in Figure 6.

The block <<wrapper>> Component recipe is made possible by the fact that UML2 Components do not "steal ownership" of UML Classes. Therefore the block wrapper Component can graphically contain many Class elements without affecting the underlying package and model structure. [9]

The block <<wrapper>> Component may also be used to logically wrap any additional reverse-engineered

8 The real-world block is considered an abstraction; it is not represented at the atomic level, it is afforded a geometrical boundary and a name and identity, and although bits of paint may fall off it during its lifetime it preserves its identity.

9 The resulting Realizations can be used to trace which elements are wrapped by a given block wrapper Component.

helper Classes used by the implementor block, to encapsulate a *logical class collaboration*. This strategy emphasises the nature of the <<wrapper>> as the boundary of a "systems machine", the contents of which are simulated by collaborating software classes. Since <<wrapper>> Components do not steal ownership, helper Classes may participate in any number of <<wrapper>>s.

The block <<wrapper>> Component may be further used as the container for tagged systems engineering concepts and development data to supplement reverse-engineered information, and/or that specified on a SysML Block. UML Comments stereotyped according to systems engineering document types may be organised this way. In this recipe, the block <<wrapper>> has the most immediate relationship to the real-world block it models, and SysML Blocks and software engineering Classes are subservient to it, they serve to achieve the goals specified via the <<wrapper>> and its contained systems engineering <<design>> Comments.

In addition, a *<<part wrapper>> Component* may be used in *block wrapper class diagrams* to logically and graphically contain information and elements related to the specific use of a block as a part within an owning block's context. In this case the <<part wrapper>> is contained in the <<wrapper>> of the Block that owns the part. This is the usual case for reuse of blocks as parts of other blocks, when the reuse context is not preempted. In other cases, a particular <<block>> Class or SysML Block may never be intended for use outside a preempted context, in which case its <<wrapper>> may be used (once only) and contained within its owning block, and no <<part wrapper>> is required, because the part only ever has one context. This subtle distinction will be illustrated throughout the model diagrams following.

In practice – once one knows the rules – both block <<wrappers> and <<part wrappers>> are easy to use, and provide enormous graphical assistance when constructing complex systems diagrams, as they help quarantine regions within the diagram from each graphically, and help one to build progressively graphically contained hierarchical systems, making diagramming more robust, and they provide for very useful navigation points using Magicdraw UML's hyperlink facility.[10].

The block <<wrapper>> Component also appears in this work in UML Composite structure diagrams and SysML Internal Block Diagrams (IBDs). This recipe provides a convenient container for Comments that refer to parts of the wrapped block, and it provides an additional context for each block system, so that relationships to other elements in the entire system can by shown using class diagram notation outside the block <<wrapper>> Component and its wrapped block (which is shown as a composite structure). Additionally, having the block <<wrapper>> Component provides for an additional navigation point to the wrapped block class diagram. This unusual recipe has proved very convenient and powerful, both graphically, and for organising systems engineering Comments and models. Diagram frames are not used here in UML Composite Structure Diagrams and SysML IBDs, as they prevent reference to other blocks in the system.

---

10 The wrapper Component practise is highly recommended, and the author now uses this recipe thoughout all of his UML-based analysis and software engineering design work, as will be illustrated in future papers.

3.4 The BlockModel/BlockPackage engineering recipe

Figure 4 also illustrates a recipe for collecting all systems engineering elements per real-world abstracted block in a <<BlockModel>>, together with its block <<wrapper>> Component, information on systems engineering artefacts (such as design manuals, maintenance data, etc.), images[11], and a contained Model for collecting instance specifications that may be used to define the *part-specific defaults*[12] for (re)configuration of parts of the Block (the values of which will override any class-level (static) defaults of the physical values of the Block type of the part Property).

The <<BlockModel>> is complemented by a software engineering <<BlockPackage>>, which contains its related interface-implementor Java pair, and other software engineering artifacts. This recipe has been found to afford enormous organisational and graphical convenience, and it is being progressively introduced into the Bragg NBI models[13]. Each real-world block is well worth a relatively cheap UML Model/Package pair.

4 Modelling examples: the OPAL neutron beam instruments

Examples of applications of the modelling recipe to some NBIs of the OPAL research reactor[1] are now presented. The detailed embedded UML Comments and figure captions are the primary explanation, a brief overview is provided here.

Figure 5 shows the “anatomy” of a fictitious neutron bunker conditioning bunker as a UML2 Composite Structure Diagram with custom <<flowport>> notation. The Comments serve to decribe the elements of the diagram, conventions, and the diagramming recipe (not an NBI). Figure 6 shows the same fictitious conditioning bunker as *a wrapped block class diagram (*equivalent to a SysML Block Definition Diagram with additional use of Component wrappers). Please note the use also of <<part wrapper>> Components and the way block classes are organised to reflect the beamline topology and reflect their usage context as part properties.

Figure 7 shows the top-level UML2 Composite Structure Diagram of the Platypus reflectometer [1], including <<design>> and <<proposal>> Comments elicited from the original proposal document and design manual. The high level bunker assembly containing the neutron beam choppers (which chop the beam into pulses for this *Time-Of-Flight (TOF)* machine) is shown in Figure 8. Note the relationship of the structured block to other “surrounding” blocks in the system, and that one component is both inside and outside the bunker. The same bunker in shown as a wrapped block class diagram in Figure 9. Note the use of both

11 Magicdraw UML now supports images diagrammatically, and this author would like to cast a vote to the OMG for Images with URL as true UML elements that can be contained and reused like UML Comments.

12 Consider a simple block with only physical values (no parts) and class-level (static) default values for each value. Consider then a structured block with a part typed by that simple block that (re)configures the part's physical values (only) with defaults specific to the part (usage context), and unique to the context of the owning structured block, which defaults shall be called here *part-specific defaults*. Such *part-specific defaults* offer the simplest form of progressive reconfiguration of parts in a progressive usage context, and are always to be applied to top-level (not deeply nested) parts within a Block, and thus do not require *property-specific (sub)types* of the part's block type.

13 Some of the diagrams here show blocks not yet employing the BlockModel/BlockPackage system.

<<part wrapper>> Components for reused block types, and the use of block <<wrapper>> Components for blocks that are particular to the Platypus NBI. Such diagrams are considered expert "software engineering views", and need not usually be used in discussions of the system with NBI scientists.

In Figure 10, Figure 11, and Figure 12 the reader is taken through some of the assemblies of a monochromation system, which is shared with variation by the OPAL diffractometers such as Echidna (High Resolution Powder Diffractometer), Wombat (High Intensity Powder Diffractometer), and Kowari (Residual Stress Diffractometerr) [1]. The entire monochromation system down to the motorised stages is shown as a wrapped block class diagram in Figure 13[14]. The motors of the motion stages are not shown (they are featured in other diagrams for the reusable motion-stage blocks, and will be the topic of a future paper).

5 The Bragg NBI ModelServer

The NBI models thus engineered under PBSE are incorporated into a Java+XML **ModelServer** system, which acts as a distributed, multi-client, control system façade[15] that integrates low-level device interactions (via a real or simulated control system) into high-level instrument components that support recursive control and derived, controllable, physical subsystem values. The ModelServer system will form the subject of a dedicated following paper, and thus will only briefly be introduced here.

The **`au.gov.ansto.bragg.base`** packages used by the **Bragg NBI ModelServer** system are not restricted to NBIs, or even to scientific instrument modelling. They provide support for a reusable Boundary – Control - Entity – Database architecture, as well as standardised logging and debugging channels, and encapsulated introspection/reflection on entity beans, to promote consistency across software systems of the Bragg Institute.

The **`au.gov.ansto.bragg.syseng`** packages of the **Bragg NBI ModelServer** system isolate the systems engineering aspects. They provide support for:

1. SysML-like "deep/rich" physical values with metadata such as units and ranges (including support for Java5 generics).
2. interpretation of physical instrument components as SysML-like blocks with simple attributes, rich physical values, references, parts, and ports, and corresponding filtered introspection support.
3. integration of low-level sensor-actuator devices with high-level physical instrument blocks (subsystems, assemblies, subassemblies, and unit/atomic blocks driven by real devices).
4. injection of real or simulated control via a configurable controller factory.

Thus, the Bragg Institute's **`syseng`** packages are intended as a target for forward-engineered instrument systems, although to date a combination of forward engineering of stubs and hand coding has been used. (See also below for experiences with EMF for generation of complete Java components.)

14 Readers are invited to consider the complexity handled by this one diagram, which is only one logical stage, of one instrument, being 1 of 9 at the OPAL facility. The block <<wrapper>> Component is your good friend for such work.

15 By comparison with the GoF [9]façade pattern the ModelServer hides the specifics of the low-level device control system so that clients may interact with a uniform API in terms of physical components and derived logical channels.

A range of ModelClients are being developed: a diagnostic, multi-column SWT TableTree desktop client for monitoring and control; a Java3D AnimatedModelMonitor; a remote WebModelMonitor; and an event logging client. An RMI network partition supports remote clients with distributed notification and callback subscription. Progress is being made towards a complete instrument simulation and control environment, including remote client support, multi-client support, animation using a Java3D client, and geometry specification directly within the UML2/SysML model.

The ModelServer is designed to integrate well with any injected device-level control system (or simulation) and to offer value-adding services to OPAL's GumTree [7] integrated scientific workbench.

5.1 Encapsulation of blocks in the Bragg NBI ModelServer

The primary domain objects of the NBI ModelServer are instances of hierarchical NBIs modelled as systems engineering blocks, starting from high-level *logical stages* of the instrument through the physical assemblies and subassemblies, down to the lowest-level indivisible "atomic" blocks, with values bound to low-level controlled devices. The ModelServer explicitly encapsulates blocks in filtered, introspected form, exposing instrument blocks to clients as encapsulated SysML-like form (see also Figure 14):

1. *(simple) attributes:* obtained by filtered introspection on the properties of the Block, which is in turn a *Bragg EntityBean (a.k.a. BraggBean)*. These are the non-physical attributes associated with enterprise software engineering, and are not usually of interest during systems operation.
2. *(rich) physical values:* representing quantities with metadata such as units, physical dimensions, and ranges: these may be further categorised as:
    - *controllable:* and thus associated with a sensor-actuator device pair (in which case a 'set' is a control command redirected through a simulated or real device controller obtained from an injected controller factory). These can be in turn sub-categorised as:
        - *directly controlled:* bound to a single low-level logical device, such as the encoded rotation angle of a rotating motions stage actuated by one motor, which receives a "logical" set command as an angle value that is translated by a given control system to motor steps using a specific driver.
        - *derived:* controlled by distribution and/or delegation of the control request to multiple low-level logical device (or to lower level derived, controllable values), orchestrated by recursive control [4], such as setting the collimation length of a neutron beam by coordinating multiple optical elements, subject to a complex computation including machine geometry and optical principles.
    - *readable:* encoded, however not driven (such as the count value of a beam monitor).
    - *editable:* a non-encoded physical variable of the machine, such as the horizontal position along the beam line of the moveable slits of the OPAL powder diffractometers (Echidna

(HRPD) and Wombat (HIPD) [1]), which are fixed by screws. The value may be edited directly in the model without going through the control system.

- *constant:* such as fixed geometrical dimensions of the physical blocks, configurable only through Java defaults, or by loading from an instrument XML instance file.

3. *parts:* SysML-like part Properties that are typed by blocks.
4. *references:* typed by blocks, seldom used, although they can provide useful navigation paths through a model in a client using hyperlinks to the referenced block.
5. *flowports*: SysML-like <<flowport>>s with a <<flow>> type, direction, and descriptive name.

5.2 SWT TableTree ModelClient

The primary visual client for the NBI ModelServer is a multi-column SWT TableTree client, which provides a hierarchical view of an instrument, from its top-level *logical stages* through its physical assemblies and sub-assemblies down to its physical unit/atomic blocks, the values of which are driven by low-level logical devices. The TableTree client's structure, values and state are completely generated from and bound to the underlying model instance using introspection and callback subscription, and thus it is a debugger-like diagnostic of the model instance within the ModelServer.

The TableTree ModelClient is shown in Figure 14 for the Platypus neutron reflectometer at OPAL [1]. The SWT ModelClient is designed for standalone operation, and for integration with the GumTree [7] scientific workbench, which is built on the Eclipse Rich Client Platform (RCP)[16].

The architecture explicitly exposes the model instance encapsulated in SysML-like fashion; each block node in the TableTree contains group nodes collecting block features to directly reflect the filtered introspection. It is the underlying architecture that encapsulates the system introspectively in a SysML-like fashion, the client merely re-presents this form, exploiting the additional horizontal columns in the TableTree to present operational parameters for monitoring and control[17]. Such model clients can and should be simultaneously “rich and thin”.

6 Reverse- and forward-engineering of UML2 and SysML models

In practice, forward-engineering of completely functioning Java components for the **NBI ModelServer** system from UML2 models proved difficult. It should be noted, however, that the PBSE UML2 models proved of enormous value even without achieving this ambitious aim. Some of the hurdles included:

1. errors in the modelling of UML2 composite structures and ports in Magicdraw UML 11.0 to 12.1 (now fixed in Magicdraw 12.5[18])
2. difficulties in specifying and generating sufficiently complex behaviours for operations

16 http://www.eclipse.org/rcp

17 The author Dr D. Kelly and his colleague Dr N. Hauser were pleased to observe the serendipitous correspondence of structure and names from the Bragg institute system to the SysML specification on first reading it in 2006.

18 personal communciation, Nerijus Jankevicius.

3. difficulties in handling mapping of Java5 generics from UML to the templated target framework
4. difficulties in specifying bindings between low-level logical devices and physical components.

Instead - with NBI commissioning immanent - it was decided to focus on forward-engineering of Java stubs from the UML2 models using Magicdraw UML's basic generation facilities, and consequent reverse-engineering of completed Java classes (after hand-coding to complete the generated Java stubs) into UML Class models, which - when combined with the block wrapper Component strategy proved quite practical.

7 Experience with the Eclipse Modelling Framework (EMF)

A systematic investigation of EMF as a candidate for forward-engineering NBI models to Java was undertaken, with mixed results. While the in-built support for XML Schemas corresponding to the generated Java proved very useful, the following aspects of EMF (as assessed in late 2006) proved a significant hurdle:

1. EMF did not explicitly support ports, which given the PBSE context was prohibitive (ports were handled only as properties, so a degree of intervention in the generated Java was required to achieve port-awareness, which proved impractical).
2. EMF did not support connections between ports of parts or between parts, which completely prohibited its use for PBSE.
3. EMF did not explicitly support deep values with units and metadata, which given the physical systems engineering context was prohibitive.
4. EMF did not (yet) support Java5 generics, and thus was not easily combined with the Bragg syseng packages.

A decision was made to abandon EMF in favour of more pragmatic software and systems engineering techniques to afford timely commissioning of software for the OPAL NBI program. A reassessment of EMF for PBSE may be performed throughout 2007/2008.[19]

8 Towards Port-Based Systems Engineering with the Magicdraw UML SysML plugin

Whereas the developmental Magicdraw UML SysML1.0 plugin was missing some features crucial for practical systems engineering of scientific instruments like NBIs , the immanent SysML1.1 plugin includes many major improvements[20], and is being assessed by this author for The Bragg Institute by application to the same NBIs modelled previously here in UML2 with custom stereotypes (see a simplied SysML Internal Block Diagram for the Platyus reflectometer in Figure 15).

Initial trials are proving very promising; in particular the flowport notation reads very well. The ability to assign and display default physical values specific to a part property (that is typed by a Block) within its unique owning Block's context makes inclusion of systems engineering data, and progressive configuration

19A proposal has been done on the Eclipse Modeling Project about the creation of a SysML Metamodel as a new component (like the UML2 component). pers.communication, SysML Forum, Raphaël Faudou 31 May 2007

20 Personal communciation, Nerijus Jankevicius.

of complex assemblies, much easier; using tagged values for such in UML2 was never a satisfactory solution.

There is, however, a real need for *property-specific type* support to afford flexible reconfiguration of deeply nested parts in complex assemblies, a case which does occur in practice (consider the complex monochromation assemblies in Figure 13 reused in three NBIs, each with its own device binding data and other variations). In fact explicit subclasses per part (as opposed to anonymous subclasses per part) created just to carry such deeply nested configuration data are used in the ModelServer Java framework; and an injectable *part configuration factory* approach is being implemented to manage this case, and as a target for forward-engineering from property-specific type notation from SysML models.

Without a facility to forward-engineer SysML blocks to a framework, such a SysML plugin offers only improved modelling and notational support (and thus better communication with stakeholders). The author is thus commencing development of a further plugin for the Magicdraw UML tool for forward-engineering of blocks to the Bragg ModelServer framework for generation of “executable” (simulatable, animatable, controllable) blocks for a wide range of systems engineering and scientific instrument applications.

The block <<wrapper>> Component strategy will assist the migration to (inclusion of) SysML blocks in the existing NBI models, and the binding of SysML blocks to forward- and reverse-engineered Java block counterparts.

9 Conclusion

This has been a very demanding and ambitious body of work, far more difficult than “regular software engineering” and in hindsight one might well have categorised the entire UML-driven instrument engineering strategy as “high risk”, given the state of the tools chosen at commencement of the project in Oct 2006.. Whereas the Java IDEs used have coped very well with this scale of modelling across an entire neutron beam facility, the performance of the Magicdraw UML tool, and the memory and the CPU resources of the desktop computers used, were pushed to their limits[21].

The complexity of the interrelationships between the UML models and the Java/XML ModelServer framework is significant. Systems engineering of complex real-world instruments with UML is not an approach that should be undertaken lightly by those who are not already very experienced with UML on a wide range of software engineering and systems engineering tasks, including experience with many established, well tested, non-UML-based systems engineering and animation tools.

However, enormous progress has been made on all fronts during the period Oct 2006 – May 2007 when this work was performed. The Magicdraw UML tool is far more stable, usable, and conforming to the UML2 specification now than when this project was commenced. And the SysML effort is providing a common language and forum for communicating known solutions and distilling known problems in UML-based

21 My patience with the robustness of the (then) Magicdraw UML tool under containment of large NBI models, and my bosses' patience with me and my strategy, were likewise also often pushed to their limits and beyond !

systems engineering.

The difficulties presented by this scale of systems engineering with UML were mitigated by the modelling and diagramming recipes presented here, such as: the use of notation to distinguish flowports from controlports and dataports; the use of wrapper Components for powerful graphical and logical grouping; the use of text elicited from (parsed from) technical documents and design manuals directly into the models and diagrams. Above all, it is crucial that one has the “grounding force” of a real-world, controlled, physical instrument or machine at hand. Systems engineering with UML/SysML cannot be performed in a vacuum.

Acknowledgements

Special thanks are due to Nick Hauser for supporting this ambitious application of model-driven development technologies “through thick and thin”, and for his ongoing interest in UML, SysML and graphical software engineering. Thanks to the Bragg Institute and ANSTO for financial and professional support throughout this work.

Thanks to Bran Selic for clarifications of aspects of the UML2 metamodel concerning composite structures, ports, and parts. Thanks to Robert Burkhardt for clarifications of aspects of the SysML metamodel regarding Block, Block Property, and Value Property, property-specific types, and defaults.

Thanks to Tony Lam for discussions on integration of the ModelServer with GumTree, and for sharing experiences with the SWT TableTree widget and networked SICS communications. Thanks to Ferdi Franceschini at OPAL for assistance with the SICS control system server during live trials. Thanks to Ron Nelson for many patient discussions of the Bragg NBI modelling framework and ModelServer system.

Thanks to No Magic Inc. for bugfixes, and for including many suggested corrections and improvements to Magicdraw UML, and especially to Nerijus Jankevicius and the Magicdraw support team. Many thanks to Daniel Brookshier for useful discussions on architecture and conformance.

And a special thanks to Nick Walker of the DESY particle physics and synchrotron institute in Hamburg, Germany, for introducing me to model-driven development, beam instrument modelling, and simulation with the Unified Modelling Languge (UML) one decade ago.

References

[1] S. J. Kennedy (2006) Construction of the neutron beam facility at Australia's OPAL research reactor. Physica B 385-386, 949-954

[2] OMG (2007) Unified Modeling Language: Superstructure Version 2.1.1, formal/2007-02-03, Object Management Group (OMG)

[3] OMG (2006) Systems Modeling Language (OMG SysML): Final Adopted Specification ptc/06-05-04. Object Management Group (OMG)

[4] Selic B (1997) An Architectural Pattern for Real-Time Control Software. In Pattern languages of program design 3. Addison-Wesley Software Pattern Series, pp 147–161, 1997, ISBN:0-201-31011-2.

[5] Austin C (2004) J2SE 5.0 in a Nutshell. http://java.sun.com/developer/technicalArticles/releases/j2se15/

[6] Eclipse Modelling Framework (EMF). http://www.eclipse.org/modeling/emf/

[7] Lam T, Hauser N, Götz A, Hathaway P, Franceschini F, Rayner H (2006) GumTree-An integrated scientific experiment environment. Physica B 385-386, 1330-1332

[8] Koennecke M, SICS: SINQ Instrument Control Software control system, http://lns00.psi.ch/sics/design/sics.html

[9] Gamma E, Helm R, Johnson R, Vlissides J (1994) Design Patterns: Elements of Reusable Object-Oriented Software. (ISBN 0-201-63361-2) Addison-Wesley.

Figures

drum assembly (which is partly inside and partly outside of the fixed monochromator shield assembly), and the beam passes through both “fixed” and “moveable” parts. The use of flowports helps to organise the model according to the logic of the beam. Note the difficulty in reflecting the horizontal beam path concisely. Note also the lack of direction on the typed “rotates with” and “is mounted inside” connections compared with the directed association names.

Figure 11 Model: UML2 composite structure diagram of the monochromator assembly, corresponding roughly to the view from above (the UML diagram can only at best indicate topology, not geometry).

Figure 12 Model: UML2 composite structure diagram of the monochromator stage assembly with motorised goniometer rotation, tilt, and translation stages, which are driven by encoded devices. Although named after the motorised, controlled values, these are still only the physical blocks. In fact, the upper motion stages will move (they rotate) when the rotating device of the lowest stage (mom) is activated, even though the controlled variables of the upper stages are not even driven. This illustrates an important difference between a low-level logical device view and a physical block view with geometry and relative assembly.

Figure 13 Model: wrapped block class diagram (software engineering view) for the entire monochromation logical stage, including deeply nested motorised motion stages of the monochromator's goniometer. Only block <<wrapper>> Components throughout (no <<part wrapper>> Components are used), since all blocks are designed for this monochromation context only, no reuse in other assemblies is intended, except for the reused motion stage blocks, which are not yet separately wrapped here. To include content specific to each motion stage part they would need to be wrapped in individual <<part wrapper>> Components.

Figure 14 SWT TableTree ModelClient showing an instance of the Platypus reflectometer model and SysML-like node groups: attributes, values, parts, ports. (Controlled values are not synched here with the live control system.)

Figure 15 Simplified SysML version of the Platypus neutron reflectometer model, with SysML flowport notation for neutron beam and air flows, control ports in UML2 notation, rich values with physical units, and part-specific default values assigned according to the usage context. Magicdraw UML's SysML1.1beta plugin was used.