The Style Guide to xtUML Modeling

Modified Date: November 10, 2011
Document Revision: 1.1
Project ID: xtUML Guide

Direct comments, questions to the author(s) listed below:
Bill Chown, 503 685 1537, at Bill_Chown@mentor.com
Dean McArthur, 613-963-1112 at Dean_McArthur@mentor.com

This document contains unpublished information that is confidential and proprietary to Mentor Graphics Corporation, 8005 SW Boeckman Road, Wilsonville, Oregon 97070. It is supplied for identification, maintenance, evaluation, engineering, and inspection purposes only, and shall not be duplicated or disclosed without prior written permission from an authorized representative of Mentor Graphics. This document and any other confidential information shall not be released to any third party without a valid Confidential Information Exchange agreement signed by the third party and an authorized Mentor Graphics representative. In accepting this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use of this information.

Table of Contents

Foreword

From determining the hardware/software partition to meeting performance and cost objectives, the job of building systems has never been easy, and with ever-increasing demand for more functionality packed into smaller spaces consuming less power, building complex systems is unquestionably becoming more challenging every day.   Add to this the desire to shrink development cycles and reduce the overall cost of the system, and you have an acute need to raise the level of abstraction and reduce unnecessary miscommunication between hardware and software teams.

Executable and Translatable UML (xtUML) accelerates the development of such complex real-time embedded systems.   xtUML is a proven, well defined and automated methodology utilizing the UML notation.   xtUML is based on an object-oriented approach that has been effectively used in thousands of real-time software and system projects, over several years and across many industries.

The key concepts that yield the productivity of xtUML build upon three principles

- Application models capture what the application does in a clear and precise manner. Application models are fully independent of design and implementation details.
- These models are executable, providing the opportunity for early validation of application requirements.
- Implementation architectures, defined in terms of design patterns, design rules and target technologies, are incorporated into a translator that generates the code for the target system. The implementation architectures are completely independent of the applications they support.

This guide is intended for systems engineers, software developers, managers and supervisors involved in the analysis and development of such systems.

The guide has been written to set out the application and methodology to achieve effective results with xtUML, and help new users quickly become productive. In this context, the supporting tools are not enough, and effective teams must have a process and methodology that leads to the desired result.

This guide will introduce each of the essential stages in the development flow, the approaches to comprehending the needs, tradeoffs and opportunities presented, and the complementary capabilities of tools and processes that are derived from best practices of many experienced users.

Contributions to this Guide

Of course, a guide such as this has many sources of expertise and authority to which we refer, and from whom we draw in offering this compilation of best practices.

Executable UML by Mellor-Balcer

In 2002, Stephen Mellor and Marc Balcer published ***Executable UML : A Foundation For Model-Driven Architecture***, which has become the definitive reference on Executable UML (xtUML). To complement this material, this Overview endeavors to apply the principles described by Mellor and Balcer in a prescriptive step-by-step approach that will assist development teams in adopting xtUML. At its Core, the xtUML Methodology employs four phases: Analysis Modeling, Executable Modeling, Model Verification and Model Compilation. Each of these phases is discussed in general terms, and a recipe provided that will enable practitioners to quickly become effective xtUML modelers.

BridgePoint documentation

The BridgePoint tool includes reference and user documentation that addresses tool-specific features, details of use of the many features, and examples of key attributes. The reader is referred to this source in addition to the methodology best practices in this Style Guide.

Additional Contributions

Other contributions come from xtUML users and practitioners in a variety of fields, and their individual and collective expertise is gratefully acknowledged.

Introduction

Several concepts and characteristics are essential to the understanding and effective application of xtUML.

This guide addresses selected major design flow steps – requirements gathering, analysis, partitioning, design, test, generation, and associated management processes required. It places these steps into the context of the four xtUML phases: Analysis Modeling, Executable Modeling, Model Verification and Model Compilation.

Analysis Modeling

Analysis Modeling is the step that moves the project from the requirements gathering stage to being ready to begin development of the Executable Models.

Requirements: At the beginning of a system project, it is common for the architecture teams to build a specification, usually in natural language. Using modeling can help elaborate the true meaning of those requirements, assemble a contextual environment in which that can be explored and effectively validated, and offer an ongoing vehicle in which derived requirements can be included and themselves explored.

Here we will look into using the best modeling techniques to understand the needs, set out the role of Use Case, Sequence, Communication, etc. diagrams to clarify requirements, and discuss how and where to reference requirements within the models.

We now enter an iterative phase that simultaneously refines and expands the collection of Sequence and Activity diagrams and begins linking them (formalizing them) to the emerging Component models that will contain the behavior of the design being created. In the course of this activity, we begin identifying domains of expertise, and decomposing designs into subject matter expertise domains for focused development. This stage includes ***Partition:*** defining a proposed partitioning into domains, for example hardware and software, so the two teams, with different skills, can head off in parallel, and ***Interface:*** the only thing connecting the two separate teams, heading off in parallel, each with different skills, is a hardware/software interface specification, and this is a key element of the formalism of modeling in xtUML.

Once iteration is complete, the Sequences and Activities clearly show the partitioning by Component and are ready to feed directly into the Executable Modeling step. The System and Design teams can now proceed with their down-stream tasks. The System team uses the Sequences and Activities to drive the creation of Test Bench Components and other System Level executable artifacts while the Design teams enter the Executable Modeling phase of the design, combining these artifacts at appropriate later stages.

In this stage of the model development, it is essential to remember some fundamental principles:

Build a Single Application Model: The functionality of the system can be implemented in either hardware or software. It is therefore advantageous to express the solution in a manner that is independent of the implementation. The specification should be more formal than English language text, and it should raise the level of abstraction at which the specification is expressed, which, in turn, increases visibility and communication. The specification should be agreed upon by both hardware and software teams, and the desired functioning established, for each increment, as early as possible.

Don't Model Implementation Structure: This follows directly from the above. If the application model must be translatable into either hardware or software, the modeling language must not contain elements designed to capture implementation, such as tasking or pipe-lining that tie the model to an implementation target. In other words, we need to capture the natural concurrency of the application without specifying an implementation. How can we capture the functionality of the system without specifying implementation? The trick is to separate the application from the architecture, and this is the key to a solution, as will be emphasized throughout this guide.

Executable Modeling

Once Analysis Modeling is complete, the requirements are broken down into a collection of scenarios that illustrate what happens when the system runs. Initial partitioning of the scenarios and mapping of elements into components should be done. Communication patterns and ordering of messaging between components is documented in the analysis models. These elements feed into Executable modeling that enables exploration of behavior and capabilities, test of features and functions, and validation of design requirements being implemented.

Executable Modeling takes the scenarios described previously and creates Executable Models that provide a testable solution to these requirements. This executable specification is directly derived from requirements and can be tested against the analysis scenarios in the form of Use Cases, Sequences, Activities and Communication patterns.

At this stage, the activity is evolving into the ***design*** stage, creating the detail that expands upon initial concepts and elaborates the full capabilities required.

Model Verification

An xtUML application model contains the details necessary to both execute and test applications independently of design and implementation. The model operates in a framework of defined timing rules allowing verification of timing relationships, as well as functional accuracy. Formal test cases are executed against the model to verify that application requirements have been properly addressed.

No design details or target code need be developed or added for model execution. Application model execution removes system errors early, with less effort and cost, and creates an unmistakably clear exit gate: a completed application model must execute.

At this stage, the design level ***test*** step can be performed, on the individual components, between components and even between disciplines. By continuing to maintain executable models and working to defined interfaces, teams can bring together executable models at any stage of the design elaboration.

Model Compilation

We translate the executable UML application model into an implementation by generating text in hardware and software description languages, and call the tool that executes this process a Model

Compiler. This is accomplished by a set of mapping rules that reads selected elements of the executable UML application model and produces text or code. The rules establish the mechanisms for communicating between hardware and software according to the same pattern.

Crucially, the elements to be translated into hardware or software can be selected by marking up the application model, which allows us to change the partition between hardware and software as a part of exploring the architectural solution space.

Generation in this manner focuses on the separation between the application behavior, described in the models, and implementation architecture, accommodated by the capabilities of the generation process.

The Modeling Style in a Particular Application Segment

Every application segment is going to be a little different, have its own needs and constraints, and company norms and processes. This guide sets out to present a starting point for the evolution of a company-specific set of guidelines for model construction that is complete and comprehensible across the enterprise.

Model Elements

In this portion of the guide, each of the elements contained within an xtUML model is discussed, and a step by step set of instructions is provided along with recommendations.

Packages

Packages group and organize the pieces of the model. They have no semantic other than control of visibility. Figure 1 shows a package that has been drawn in BridgePoint Builder as well as the palette tool used to create new packages.

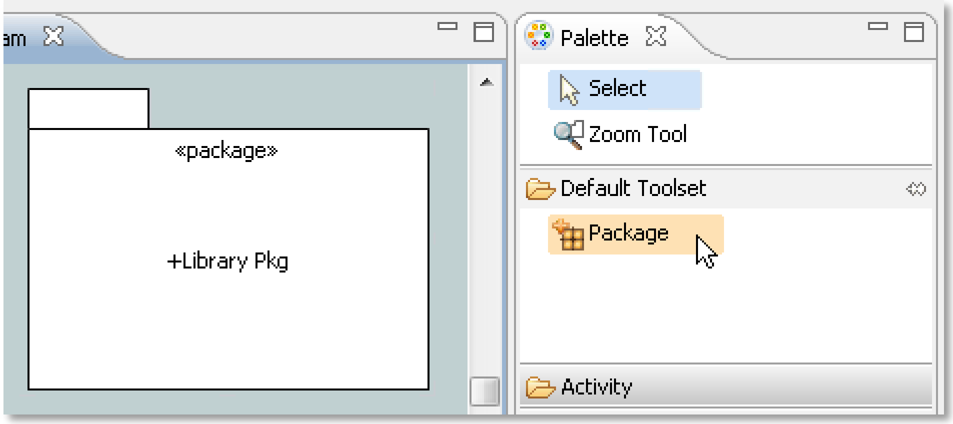


Figure 1

BridgePoint has two types of packages: generic and specialized.

Specialized packages are restricted to contain only one type of element (for example, components or data types). Specialized packages have a history in the Shlaer-Mellor modeling notation. As of BridgePoint 3.4.0, specialized packages are deprecated and generic packages are preferred.

Generic packages, called simply “Package” in the BridgePoint tool palette, can contain many different types of elements conforming to the UML 2.0 specification of what a package is and what it can be used for [1]:

> Packages are general-purpose hierarchical organizational units of UML models. They can be used for storage, access control, configuration management, and constructing libraries containing reusable model fragments.

Packages have visibility properties that enforce how the data inside the package can be accessed by elements within, nested under, or outside the package. Packages also serve as an organizer of the modeling projects. To effectively employ packages, the guidelines regarding descriptions, naming standards, and organization, which are provided next, should be considered:

Enter Descriptions

An important guideline is to "document as you go" by entering descriptive information about each package you create when you create it. All too often analysts choose not to perform this step surmising that they will circle back and add it later. This second pass to write the documentation rarely happens, and it is not surprising. The context and ideas that drove the creation of the model elements become stale or lost completely. The pressures of the later phases of the project outweigh a documentation pass. The size of the data to document often becomes so large, that the task seems insurmountable. Or, the original creator of the model elements has moved on to other work. These are just a few of the reasons that the documentation information needs to be entered as elements are created. Choosing not to do this is a disservice to yourself and your team.

Organization

There are no hard and fast rules about where packages, either for Analysis or Executable modeling, should go in the hierarchy. Related packages should be grouped, following the rules of cohesion. That is, maximize close proximity of related subject matter and minimize package interdependency.

Use Separate Packages for Libraries and System Wiring

You should create packages that serve as libraries of components. For example, create a package that is a library of components which models hardware functionality. Create another package that is a library of components that serve as the hardware abstraction layer (HAL). Create another package that is a library of components that serve as a test bench for the components elsewhere in the model. Create another package or packages that contain the application itself. Create packages to contain the interfaces that the components expose.

Then, create one or more packages that define scenarios for how the system is *wired* together.

> *Definition*: to *wire* components references together is the act of connecting a provided interface from one component reference to a matching required interface of another component reference.

Continuing the previous example, the project should have a “System Test” package with component references to a hardware component, a HAL component, and a test bench component wired together. The project should have another package “System Implementation” that includes HAL component references wired together directly with application components. The system implementation package is what is passed to the model compiler for translation into implementation code.

BridgePoint allows you to wire together components themselves as well as component references. However, direct wiring of components is not recommended.   Stick to creating component libraries, and only wiring together component references.

Let’s consider a simplified example using the case study project included in the BridgePoint help. Figure 2 shows a library package that contains two components. The component interfaces are specified, but the interfaces are intentionally left disconnected.

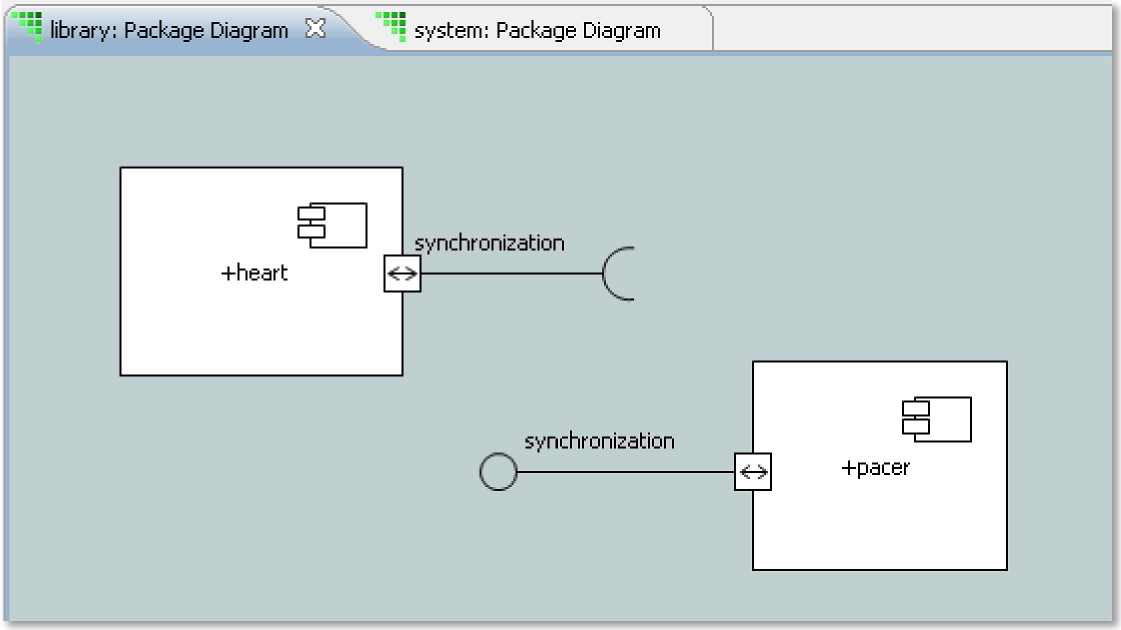


Figure 1

The wiring diagram, a package named “system” as shown in Figure 3, is where the component references from the library have the interfaces connected together.

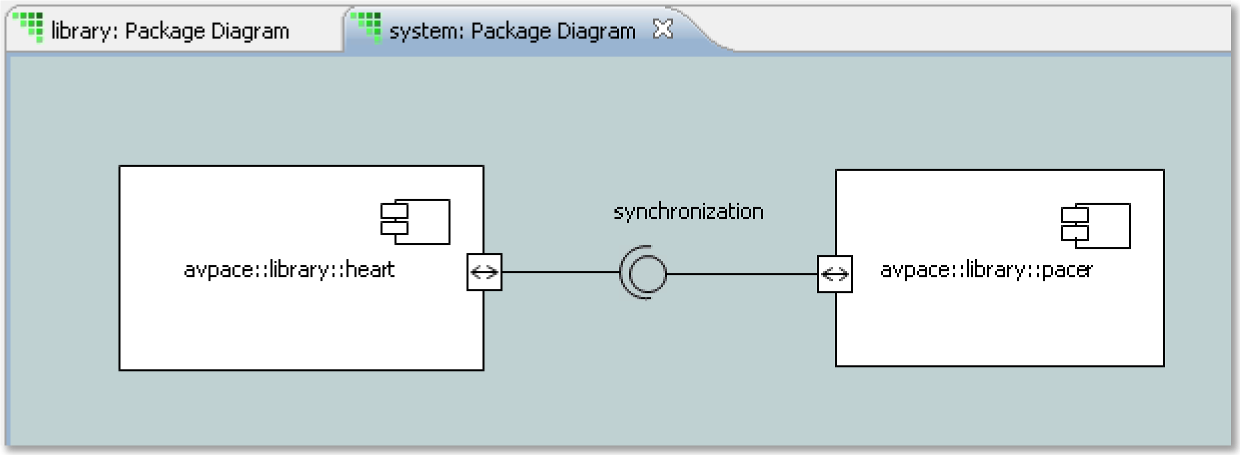


Figure 2

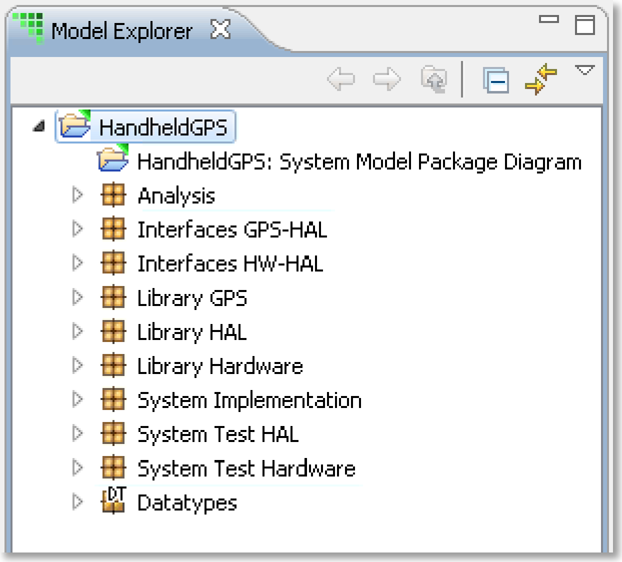
By leaving the interfaces disconnected in the library, you are free to create other components and system wirings that connect the component references from the library together in different configurations.

Naming Conventions

Since packages provide the organization of the various pieces of the model data, package names should give some indication about the data. Here are some suggestions:

- Use a meaningful name over a generic or overly-broad name

- Put the purpose of the package first so that similar packages are grouped together in Model Explorer and provide organizational information.



HandheldGPS
HandheldGPS: System Model Package Diagram
Analysis
Interfaces GPS-HAL
Interfaces HW-HAL
Library GPS
Library HAL
Library Hardware
System Implementation
System Test HAL
System Test Hardware
Datatypes

Figure 3

Visibility and Namespaces

Packages support public (the default), protected, and private visibility. On the UI canvases, the package name is preceded by an indicator to show the current visibility setting: + (public), - (private) and # (protected).

Name-spacing is supported and managed entirely by means of visibility. That is, two elements with the same name are permitted provided that they are not simultaneously public in any action language scope. More than one type found during parsing is indicated by a parse error.

Note that changing the visibility of a container has no effect on the visibility setting of a child. If a package is visible to another element, it will be descended, regardless of the visibility setting. You may wonder: “So, what is the point of marking a package as private?” It is in the opposite conclusion; if a package cannot be seen, it will not be descended.

The behavior is closely analogous to that a text based programming language. For example, can you see public members of a private class in Java? Yes, but only if you are in the same package as that class. In fact, if you are not in the same package, you can't see the class, let alone its members. Here, the visibility of the container does not affect the visibility of its members.

In the default workspace configuration with inter-project references turned off, a consequence of the rule above is that marking a top level package as private is meaningless, since there is nothing above it to be hidden from. However, if the workspace has inter-project references turned on, top level package visibility does matter because it allows you to control which packages can be seen (and hence descended in searches) from other projects.

Visibility matrix

Referred to Element is	**Referred to Element is**		
	public	protected	private
Above in same package	V	V	I
Below in package (1)	V	I	I
Peer	V	V	V
Above in sibling package (1)	V	I	I
Above in same Component	V	V (3)	I (2)
System Level	V	V (3)	I (2)

Legend: V = Visible, I = Invisible

Notes:

(1) Assumes parent package is visible according to the same rules.

(2) A private element immediately under the system or component has very limited visibility, limited only to peer elements at the declared level.

(3) Marking an element as protected in a system or component level context is meaningless since the semantics are no different from the public semantic.

Data Type

UML data types [1] are, by nature, very similar to data types in implementation languages.

BridgePoint data types are based on a two-level scheme:

- Core or Primitive data types are fundamental to the language. Core types are the base from which other data types can be defined. Model elements may be declared to be of a core type. The core types are:

void	inst<Event>
boolean	inst<Mapping>
integer	inst_ref<Mapping>
real	component_ref
string	date
unique_id	inst_ref<Timer>
inst_ref<Object>	timestamp
inst_ref_set<Object>	

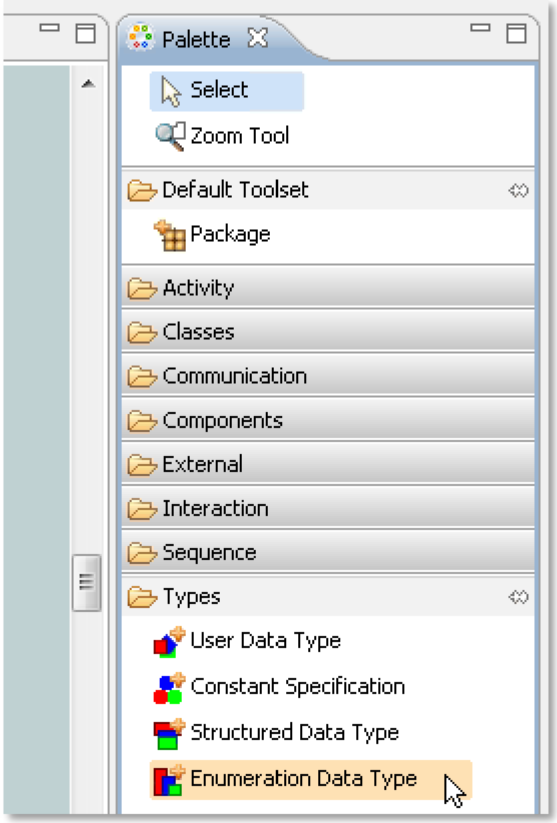
- Domain-specific data types are added by you to define and extend core types in terms of the application domain.

For additional discussion of this scheme, see [2]. The BridgePoint help (See Help > BridgePoint UML Suite Help > Reference > Using BridgePoint > Model Elements > Data Types provides detailed definitions for each of the core data types.

Steps to creating Domain-specific Data types

The same Help document defines the options to create domain-specific data types. The tool palette “Types” drawer shown in Figure 4

Figure 5 contains the domain-specific types you can add to your model.



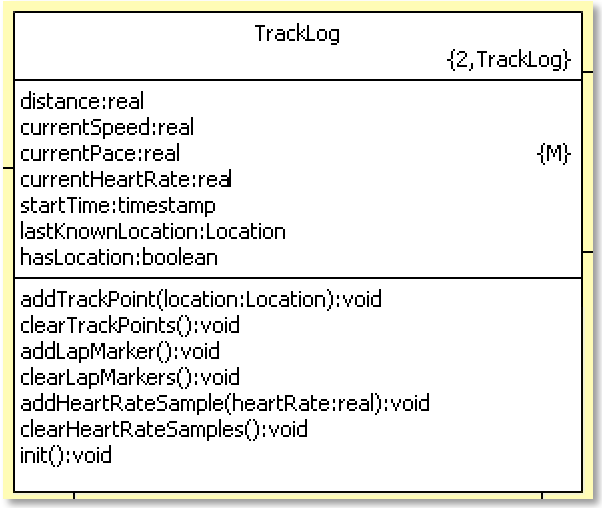


Figure 4

Figure 5

BridgePoint shows the type of elements on appropriate diagrams throughout the tool. Figure 4 shows an example from a class diagram. Attribute, parameter, and return value types are shown following their associated element.

Both core and domain-specific types are valid in all contexts where types are used. For example, it is just as valid to use a structured or enumerated data type as a class attributes type as it is to use those types for a parameter or return value of an interface message.

General Guidelines

- Document all new data types you create by writing description information in the Properties view when the type is created. This “document as you go” approach has many benefits and should not be ignored.

- Create Packages to contain domain-specific data types. Data type packages may be nested inside components if the type is specific to that component or its children. Create packages to hold data types at the system level to organize and contain types that may be used on the interfaces between components or are generally applicable to more than one component in the application.
- Create and use enumeration data types to specify lists and values. For example, the enumerated type "MatterState" with values (Solid, Liquid, Gas) is preferable to using integers with values 0, 1, and 2. Enumerated types provide contextual information and allow the parser to restrict code to legal values where, in this example, an integer would not.
- Use constants instead of embedding raw values into action language.
- Some advice from Mellor and Balcer [2] is applicable as well:

Syntax vs. Semantics. Define attribute types in terms of the meaning of the data (the semantics), not merely in the form of the data. Good models should use domain-specific types as much as possible and refrain from using the core data types if more meaning is available. Hence, defining an attribute merely as real is not as good as using a domain-specific type, such as Currency or Supply Voltage.

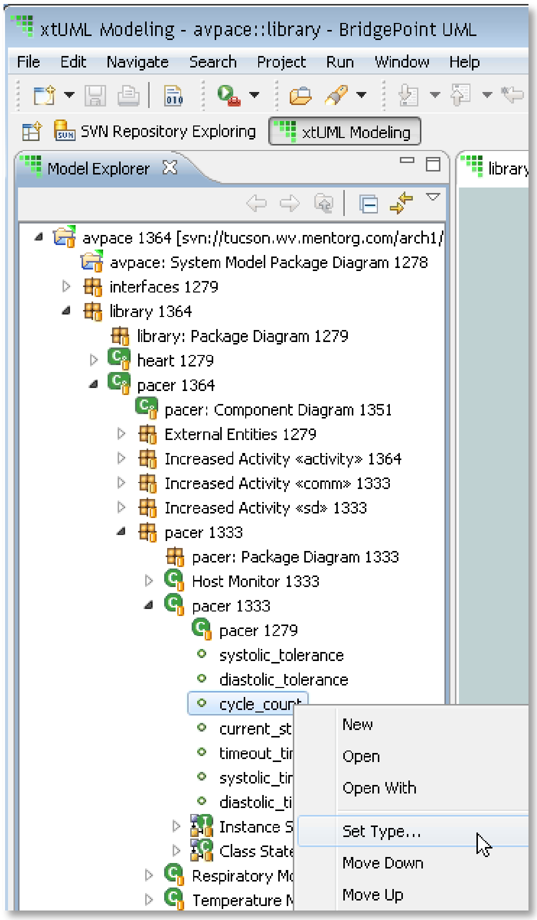
Naming Conventions

- Choose meaningful names for domain-specific types. The name of the type should provide contextual information about where it is appropriate to use. For example, don't name a domain-specific type "MyEnum" when "MatterState" is appropriate. Don't create an integer-based User Defined Type "MyShortInt" and overload its usage when two types "PostalCode" and "AreaCode" are appropriate.
- Name domain-specific types consistently. Don't name one type "MatterState" and another "supply_voltage".

Visibility

Data types are subject to the same visibility rules that govern packages and components. See the section **Packages within the Style Guide** for more information about the visibility rules.

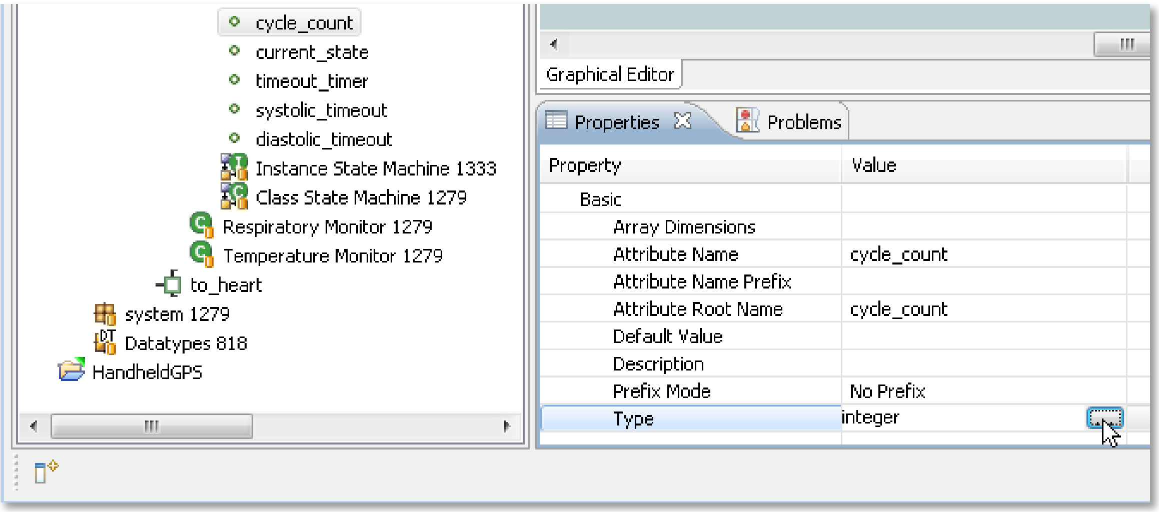
Setting Types



BridgePoint provides two ways to modify the type of a model element (for example a class attribute or interface parameter). The primary method is shown in **Error! Reference source not found.**, using the "Set Type…" action on the element context menu in the Model Explorer view.

When you select an element in the Model Explorer view, or on one of the BridgePoint diagrams, the Properties view is populated with information (including the Type) about the selected object. The Type field is modifiable here by selecting the "…" button.

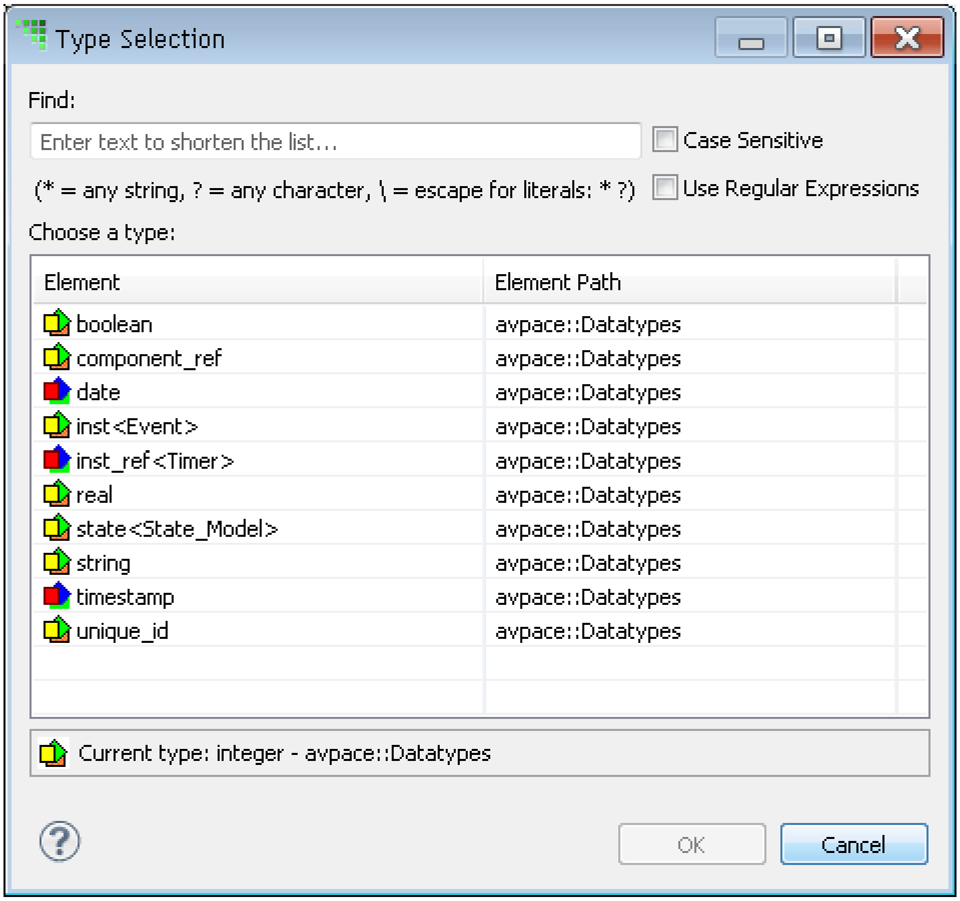
Either action may be used with the same outcome, which is to open the Type Selection wizard shown in Figure 6. This wizard shows all the types the model element may be set to. The wizard automatically



Property	Value
Basic	
Array Dimensions	
Attribute Name	cycle_count
Attribute Name Prefix	
Attribute Root Name	cycle_count
Default Value	
Description	
Prefix Mode	No Prefix
Type	integer

hides types with restricted visibilities that are invalid.

Elements must have their type set individually. Group type modification is not supported. However, the wizard automatically highlights the last type you chose. This saves you a few mouse clicks when setting a number of elements to the same time. The wizard dialog includes a "Find" field, where you can enter a regular expression to locate and restrict the available types shown in the chooser. Note, the existing data type of the selected element being modified is shown in the "Current type" field. The existing type is never shown as an available choice.



□ Case Sensitive

(* = any string, ? = any character, \ = escape for literals: * ?) □ Use Regular Expressions

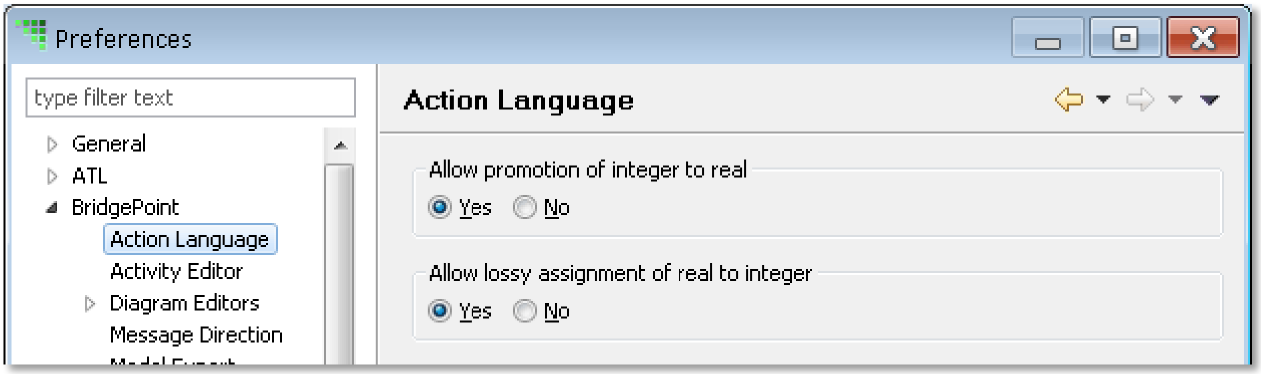
Element	Element Path
boolean	avpace::Datatypes
component_ref	avpace::Datatypes
date	avpace::Datatypes
inst<Event>	avpace::Datatypes
inst_ref<Timer>	avpace::Datatypes
real	avpace::Datatypes
state<State_Model>	avpace::Datatypes
string	avpace::Datatypes
timestamp	avpace::Datatypes
unique_id	avpace::Datatypes

Current type: integer - avpace::Datatypes

Figure 6

Data Types in Action Language

The OAL parser checks type compatibility in the action language code. It prevents you from assigning variables of mismatching types, passing variables of the wrong time to operations and interfaces, etc. When dealing with integer and real numeric values, the rigidity of type enforcement can be configured in the BridgePoint preferences as shown in Figure 7.



Allow promotion of integer to real
● Yes ○ No

Allow lossy assignment of real to integer
● Yes ○ No

Figure 7

“Allow promotion of integer to real” determines whether or not the parser will transform an integer into a real when performing mathematic or assignment operations. The default is “Yes.” If you want to enforce strict type compatibility, change this preference to “No.” Changing this setting to increase strictness automatically enforces strictness in the other preference.

“Allow lossy assignment of real to integer” determines whether or not the parser will allow assignment of a real to an integer variable or attribute with the consequence that the decimal portion of the real value will be truncated off.

Data types in OAL are handled implicitly. There is no explicit declaration of a transient with a specific type specified. To “declare” a transient to be of a specific type, simply write an assignment statement that introduces the transient and gives it a value. For example:

// Correct	// Incorrect
index = 0;	integer index;
voltageLimit = 1.5;	index = 0;
message = “Hello World”;	String message = “Hello World”;

To use an enumeration data type in OAL, the syntax is <enumerated type name>::<enumeration>. For example, using the MatterState enumerated type defined in the General Guidelines section above:

```
// Using an enumerated type
heating = true;
if ( ( currentState == MatterState::Liquid ) && heating )
    nextState = MatterState::Gas;
else
    nextState = MatterState::Solid;
end if;
```

The members of a structured data type are accessed with the "." operator, the syntax is: *<sdt variable>.<sdt member>*. Consider an example from the GPS Watch example model. Here, currentLocation is of type Location with is a structured data type with three real members (longitude, latitude, and speed):

```
select any gps from instances of GPS;
if (empty gps)
  create object instance gps of GPS;
end if;

// reset currentLocation
gps.currentLocation.longitude = 0.0;
gps.currentLocation.latitude  = 0.0;
gps.currentLocation.speed     = 0.0;
```

Constants are used directly based on the constant name. There is no scoping based on the name of the constant specification that contains the constant. The constant specification name has no semantic meaning. It is for informational and organizational purposes only and may be blank. For example:

```
// A constant specification named “Messages” with constants:
//    LISTENER_REGISTERED = Location listener registered.
//    LISTENER_UNREGISTERED = Location listener unregistered.
// exists in a package at the system level.

if ( registered )
   LOG::LogInfo(message: LISTENER_REGISTERED);
end if;
```

Data Types and Model Compilers

The BridgePoint model compilers provide a means to configure domain-specific user data types for your specific implementation needs. As with all model compiler configurations, this is done through *marks*. The marks are found in the `<project>/gen/datatype.mark` file. Marking is used to define the precision of these user data types. This is particularly useful to reduce the storage (say from 16 or 32 bits to 8 bits) of class attributes when the ranges of the attributes are known to be limited. Marking can also be used to map pointer types, specify uninitialized enumerators, and specify specific values for enumerators. There is detailed help in the BridgePoint Model Compiler User’s Guide in the “**Specifying Data Types, Precision, and Enumerators**” section. The `datatype.mark` file itself contains additional information and examples for using each of the available marks.

Data Types and Inter-Project References

Currently, BridgePoint models are created by default to be self-contained. That is, they do not refer to any modeled artifacts not defined within the current project. It is often convenient to create a separate project to act as a model library. However, in order to do that, the cross referring projects must have a common definition of the Core Data types (integer, real and so on). If this were not so, the Object Action Language parser could not confirm the compatibility of data passed between instances in different projects, and Verifier could not execute such inter-project references. Therefore, before you can reference elements in another project, both projects must have been upgraded to use common global core data types.

You can tell if a project has already been upgraded to use global data types. Click the right mouse button over the root entry of the project in Model Explorer. If the menu entry ‘Upgrade to use global model elements…’ is present then the project needs to be upgraded.

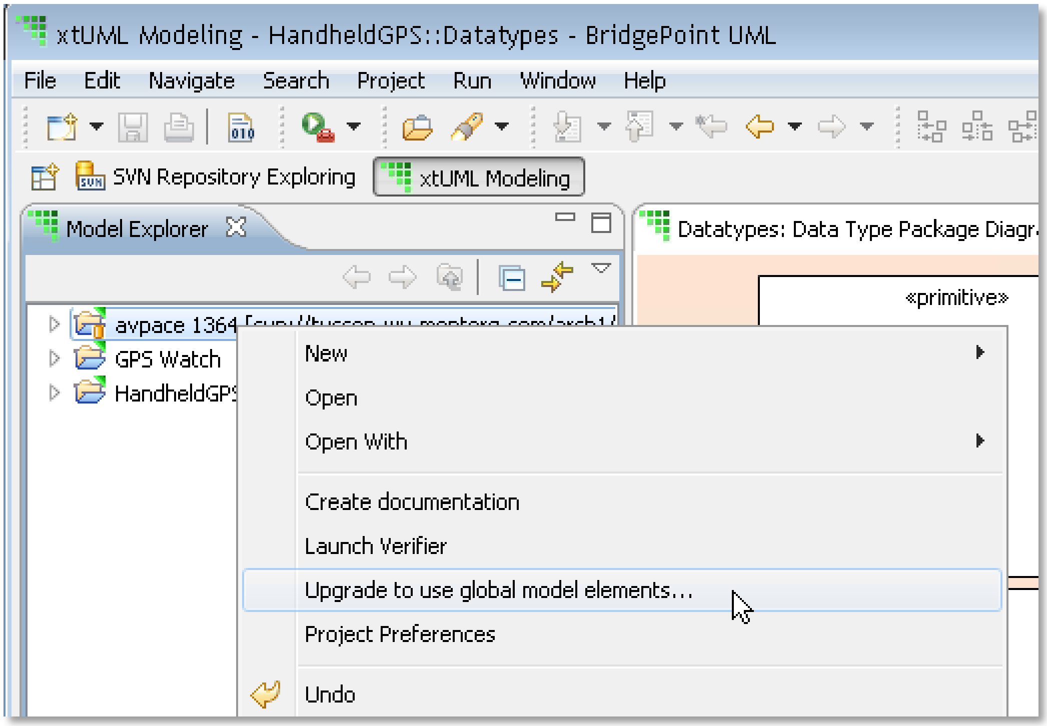
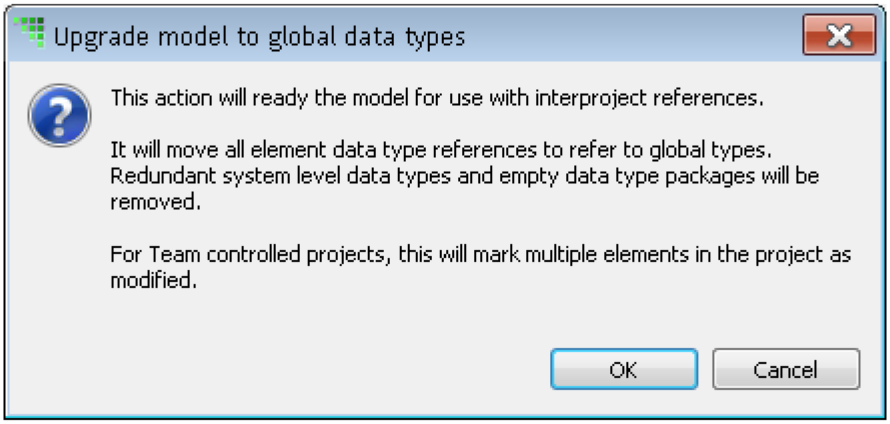


Figure 8

Choosing this menu action will present you with the dialog shown in Figure 9:


Upgrade model to global data types

This action will ready the model for use with interproject references.

It will move all element data type references to refer to global types. Redundant system level data types and empty data type packages will be removed.

For Team controlled projects, this will mark multiple elements in the project as modified.

Figure 9

The process will find all references to built-in data types in your model and replace them with references to the global equivalents. Be prepared for large parts of the model to be marked as modified if you use a Configuration Management system. However, the visible changes to your model will be hardly noticeable. The biggest change to expect is that the system level package; ‘Datatypes’ may be removed if it is found to be empty after the upgrade. This will be the case if you have never added any types of your own to this package. No user created data types will be removed.

Once this is done, you can enable inter-project references by clicking on each System Level in the Model Explorer tree and choosing Project Preferences > Inter-project References.

Application of Timing

The UML reference manual [1] defines the ability to model time in a diagram, but it does not provide a definition of what time is or semantics for how time behaves. To the reference manual, time is little more than another type of graphical element that can be drawn on a diagram.

Mellor and Balcer expand on the concept of time in *Executable UML*[2]:

> **For UML to be executable, we must have rules that define the dynamic semantics of the specification. Dynamically, each object is thought of as executing concurrently, asynchronously with respect to all others. Each object may be executing a procedure or waiting for something to happen to cause it to execute. Sequence is defined for each object separately; there is no global time and any required synchronization between objects must be modeled explicitly.**

Signals are the means by which objects communicate and synchronize with each other. They are interpreted by the receiver which executes a procedure as an effect. The procedure has its own processing which may include data access, operations and the sending of its own signals.

Executable UML goes on to define rules about signals, procedures, and data access that are implemented in BridgePoint xtUML. These rules define the execution semantics.

Time (TIM) External Entity

BridgePoint provides a built-in external entity named Time (key letters TIM) to implement access to timing and clocks.

xtUML supports two different concepts of time in the Object Action Language that provide concrete access to time in the implementation:

> *External time*: Time as known in the external world. For example, 12 October 1492, 13:25:10. The accuracy of external time is dependent on the architecture and implementation.
>
> *Internal time*: An internal system clock that measures time in "ticks". The value of a tick is dependent upon the architecture and implementation.

An implementation of the TIM EE is defined inside Model Verifier. It provides a Java implementation that runs on the host machine inside BridgePoint.

The Model Compiler provides an implementation of the TIM EE in the target language that runs on the target itself.

Detailed usage syntax for accessing external and internal time may be found in the Help System under BridgePoint UML Suite Help > Reference > OAL Reference > Date and Time.

BridgePoint does not natively support differencing of internal timestamps. However, an interesting example of how this can be achieved by extending the application is found in the "Another Example" section of the Model Verifier Java Interface document (BridgePoint UML Suite Help > Reference > Using BridgePoint > Model Verifier Java Interface).

Timers

A *timer* allows a pre-created event to be delivered at some future time. This operation starts a timer and sets it to expire after a specified number of microseconds, generating the specified event upon expiration. This can be done in a once-and-done or recurring manner.

It is important to note that the delay given specifies the <u>minimum</u> delay, not the exact delay. The model relies on the characteristics of the target architecture and the execution environment to deliver the event as soon as possible after the timer expires. In many architectures, there may be a delay between the expiration of a timer and the delivery of the event to the receiver. The xtUML execution semantics have no control over this. The execution semantics can only guarantee that the timer will never expire before the given delay has elapsed.

In addition to creating and starting a timer, you can query a timer and receive the time remaining specified in microseconds. If the timer has expired, a zero value is returned.

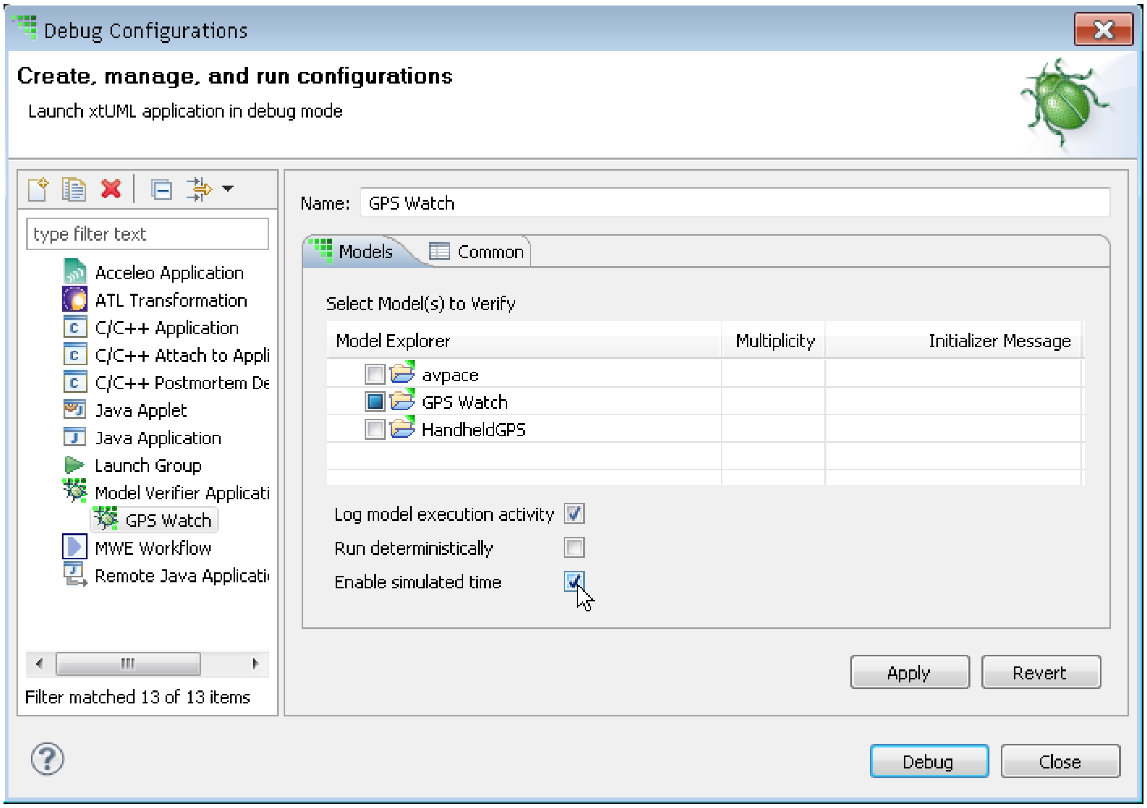
The TIM EE also supports setting a new expiration delay on an existing timer, overwriting the original expiration delay as well as adding time to the original expiration delay.

Finally, the TIM EE supports canceling a timer, which deletes the timer completely such that the event associated with the timer will never be delivered.

Detailed usage syntax for accessing external and internal time may be found in the Help System under BridgePoint UML Suite Help > Reference > OAL Reference > Timers.

Time in Model Verifier

Model Verifier supports two ways of viewing time when debugging application execution. *Wall clock time* and *Simulated time*. Each Model Verifier launch configuration can set this option as it chooses as shown in Figure :



Debug Configurations

Create, manage, and run configurations

Launch xtUML application in debug mode

Name: GPS Watch

Select Model(s) to Verify

Model Explorer	Multiplicity	Initializer Message
□ avpace		
■ GPS Watch		
□ HandheldGPS		

Log model execution activity ☑

Run deterministically □

Enable simulated time ☑

Filter matched 13 of 13 items

Figure

When executing in simulated time, when the application processes all the outstanding events on the event queue and the application is waiting for timers to expire and post new events to the event queue, the system recognizes this and advances internal time to the point where the next timer will expire. By doing this, the application runs much faster because the launch configuration avoids any delays where the application is effectively sleeping.

When executing in wall clock time, this method of advancing time is not used. All delays or periods of the application sleeping are waited out the duration of the delay in the real world on a clock on the wall.

Simulated time is especially useful for testing purposes. It allows scenarios to run faster in simulation allowing the test to cover more ground in simulation and debugging in a shorter amount of wall clock time.

“Run deterministically” is another execution preference. It controls the internal behavior of Model Verifier, and is not primarily related to time so it is not detailed here. However, it does force the application to run in simulated time to support the features it provides.

Action Language

OAL is used to define the semantics for the processing that occurs in an action. An action can be associated with any of the executable BridgePoint model elements. This section provides some guidance to use while writing OAL. For basic OAL syntax see the BridgePoint OAL Reference manual in the BridgePoint UML Suite Help. For a more detailed information about why OAL is needed in executable model see [1].

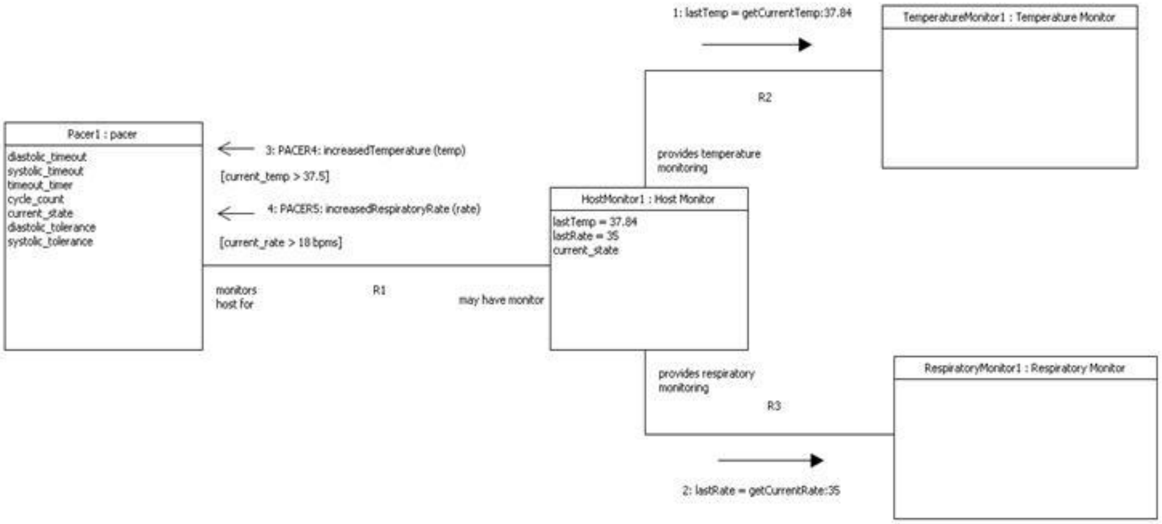
General Guidelines

- Make the OAL Readable
  - Make OAL "beautiful" by using white space and consistent formatting.
  - Modelers must be able to understand the OAL for code reviews and maintenance.
  - Add comments liberally!! The comment character in OAL is: "//"
- Leave a history if possible.
  - Add references to issue IDs (from an issue tracking system) .
  - Adding issue identifiers allows changes to be tracked back to the reason for the change.
  - When checking-in changes to your revision control system always use good comments in the check-in comment!
- When writing OAL think about another Modeler who may have to look at this OAL over a year from the point in time you are writing it. Comment the OAL and style it to make it easy for that Modeler (HINT: You may be this modeler that revisits this OAL over a year from the time it was originally written!).
- Use BridgePoint's Model Verifier to quickly execute, test. and validate OAL .
- To assign NULL to a variable use a native Bridge operation that returns null
  .

Diagrams Guide

In the documentation of requirements and the evolution of xtUML models, diagrams are extensively used to precisely describe the desired operation of the system. With this importance, we have included this section to discuss each diagram and provide step-by-step instructions on producing intelligible, informative diagrams.

Communication diagram


1: lastTemp = getCurrentTemp:37.84
TemperatureMonitor1 : Temperature Monitor
R2
Pacer1 : pacer
diastolic_timeout
systolic_timeout
timeout_timer
cycle_count
current_state
diastolic_tolerance
systolic_tolerance
3: PACER4: increasedTemperature (temp)
[current_temp > 37.5]
4: PACER5: increasedRespiratoryRate (rate)
[current_rate > 18 bpms]
provides temperature monitoring
HostMonitor1 : Host Monitor
lastTemp = 37.84
lastRate = 35
current_state
monitors
host for
R1
may have monitor
provides respiratory monitoring
R3
RespiratoryMonitor1 : Respiratory Monitor
2: lastRate = getCurrentRate:35

Use a Communication Diagram to document the details of a certain procedure and to depict the interaction of associated objects within a system. The interaction details include the associations between the objects of interest. In addition, the details include the communication that can occur between the objects. The time sequence for the communication can be captured by using sequence numbers in the message labels.

Things to Know

Participants (Component, Instance, Actor, External Entity, Class, Package)

Messages (Synchronous, Asynchronous, Return)

Links

Whe*n *to use

A Communication Diagram can be used formally or informally. In the formal case the various objects, associations, and messages will already exist. The Communication elements will be formalized against (referencing to) the existing objects and will be used to depict a procedure that occurs among the existing objects. The resulting document will aid in detailing a single procedure, while filtering out all possible procedures among the objects.

In the informal case there will be no existing objects, associations, or messages. The Communication elements will be created to document a required procedure, including the objects and their interaction. The elements can then be used to realize the objects, associations, and messages. When creating an informal document a few things must be known. The following checklist can be used to get started:

- The required procedure and its goal, what must be accomplished.
- The various objects that participate in the procedure.
- The associations among the participating objects.
- The messages between the participating objects that are called in order to achieve the procedure goal.

The resulting document is living and may change over time. When focusing on the informal case, the exact objects, associations, and messages do not have to be precise. The document will change as these elements come to realization.

A well written Communication Diagram will be short and precise at describing the desired procedure. Include only objects that participate in the procedure, and include only messages between the objects that participate in the procedure. If the procedure to be documented is large, break the procedure into sub-procedures each with its own Communication Diagram.

Formal Usage:

As stated above in the formal case the objects, associations, and messages will already exist. A Package is used to enclose communication elements and thus represents the Communication Diagram. Create the Communication Diagram at a location that makes sense given the existing elements. Place the diagram in the hierarchy of packages following an organized strategy.

Select the appropriate tool to create a participant. This will be one of the listed participants from the Elements section above. Choose the tool that will match the formal object. For instance if the content of the Communication Diagram will show the interaction of Components, then choose the Component Participant tool. Create one participant for every object that will participate in the documented procedure. For each participant created, use the formalization wizard to locate and use the appropriate object.

Now select the Link tool, and create the link(s) that are to be documented in the procedure. Create these links between each object created above. There will be a matching association, or interface satisfaction for the component case. For each link that represents an association, use the formalization wizard to locate and use the appropriate association.

Now create the necessary messages that are documented by the procedure. In most cases this should not be all of the possible messages that are between two associated objects. The diagram shall focus on the procedure to document and shall ignore any messages that are not pertinent to that procedure. Create the messages next to the created links. Multiple messages can be present for each link as long as they pertain to the procedure to be documented. For each message use the formalization wizard to locate and use the appropriate action.

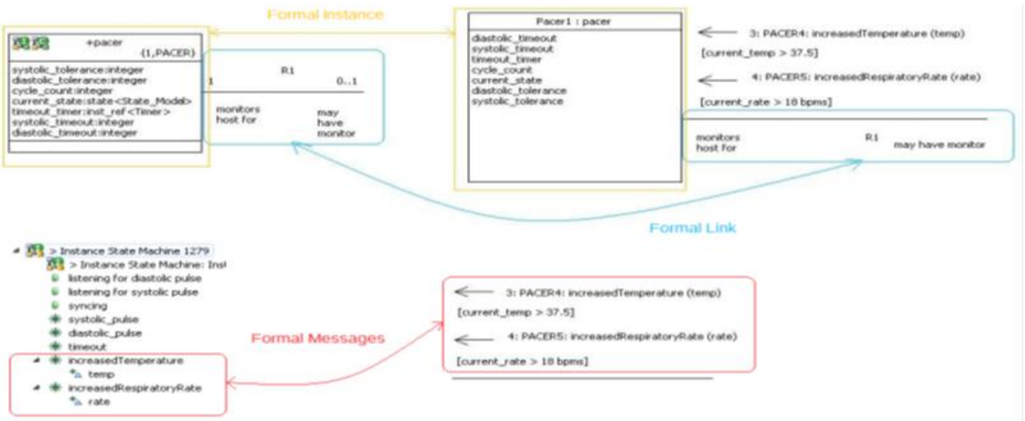


Figure 1.1 (Showing formal referencing)

Informal Usage:

When creating the Communication Diagram informally, there are no existing elements to base the procedure on. The diagram will serve to provide aid when creating the real elements. An informal diagram will start off informal but will later migrate to a formal diagram. While creating the diagram consider carefully the procedure to be documented, taking care to name the informal elements appropriately.

The mechanics of creating this diagram are identical to that of the formal case. The one step that will not be taken is using the formalization wizard.

The informal Communication Diagram elements have attributes that can be set which allow further definition. Informal elements can be configured such that enough data is present to allow fully configured realized elements. The proceeding How To section will explain how these informal attributes are used.

Conversion to formal document:

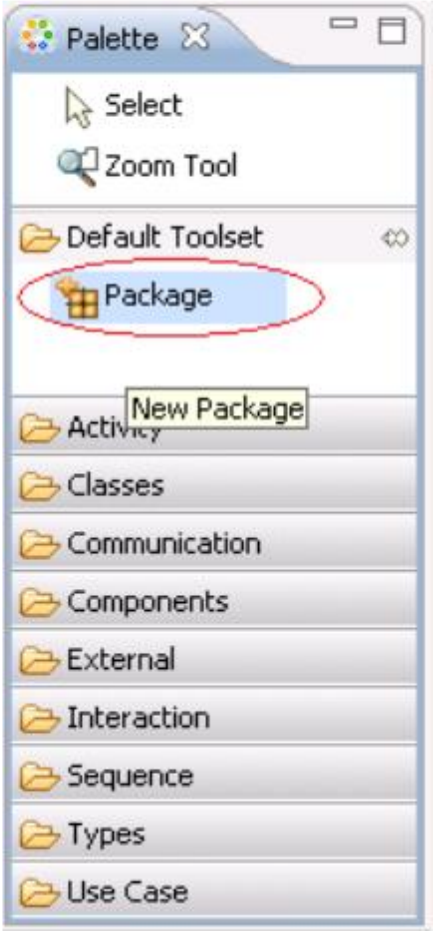
As stated above at some point all Communication Diagrams shall become formal. After an informal diagram has been agreed upon by all stake holders, it is time to realize the procedure using the following steps:

1. Create an object for each participant in the procedure. The object created depends on the participant type. The exact steps for such creation are not within the scope of this document. For example, if the participant type is component, then a matching component shall be created in a package.
2. Create an association or interface satisfaction for each link that exists in the Communication Diagram. Again this depends on the type of the participants at each end of the link.
3. Create the necessary action, i.e., operation, for each message that exists in the Communication Diagram. The destination for the actions will be the object that the participant represents at the end of the message on the diagram. The end of the message is denoted by the arrow end.
4. After all of the objects, associations, and actions have been created it is time to formalize the original document. Select each participant and use the formalization wizard to locate and use the objects created above. Repeat this step for each link and each message within the document.

Steps to creating a Communication Diagram

Diagram creation:

Once the location has been determined, locate the palette view (see Figure 1.2 below). In the palette view select the Package tool (as shown in Figure 1.2). On the editor page click the left mouse button and drag a distance. A marquee will be drawn that indicates the location and size of the graphical symbol to be created. Once the symbol is at a desired location and size let go of the left mouse button. At this point a new Package has been created. Right click on the Package graphical symbol and select “Rename” (See Figure 1.3 and 1.4 below). In the window that is opened enter a good name for the package. This name should reflect the procedure that is being captured by the Communication Diagram.



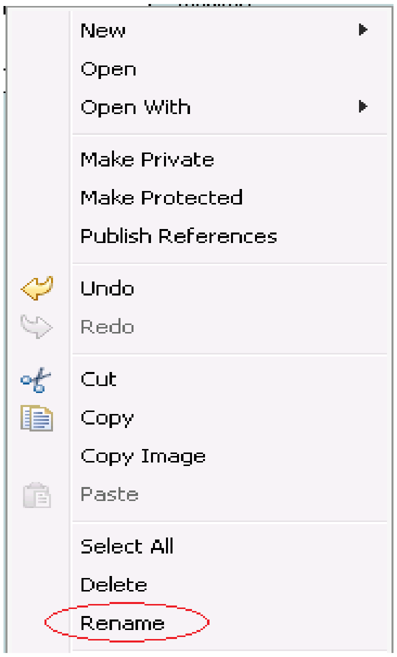
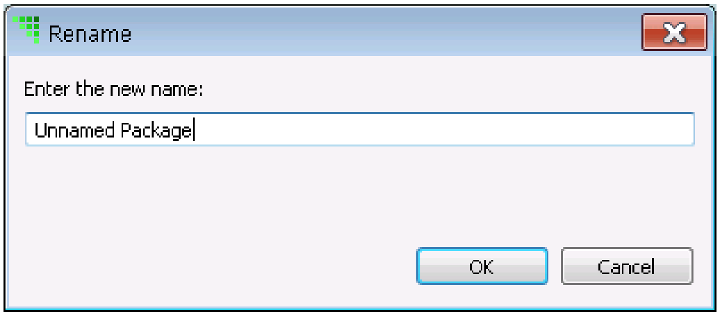


Figure 2

Figure 3

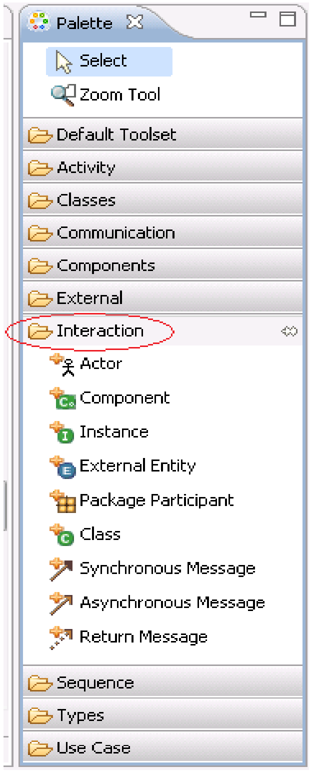


Enter the new name:

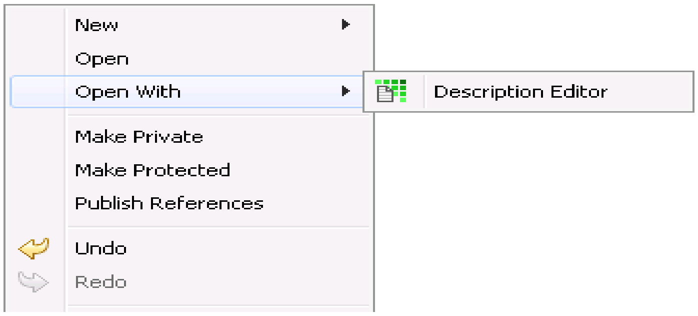
Unnamed Package

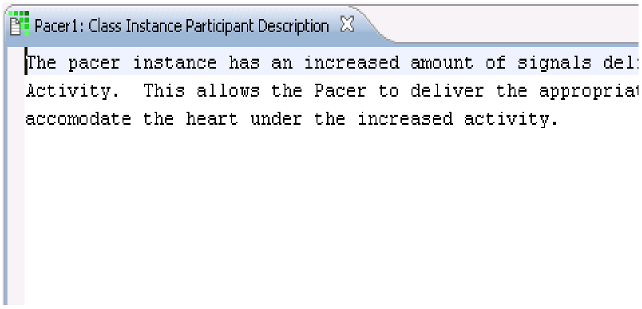
Figure 4

Participation creation:



As with diagram creation, the proper tool must be selected. As stated in the Elements section there are many participants. Select the participant tool that is of interest (see Figure 1.5 below). Once selected proceed to draw the symbol for the participant graphical element (see steps in package creation for details). At this point the newly created participant can either be formalized or customized to represent a future formal element. For either case, formal or informal, it is a good idea to enter a description for the element created. The description shall describe the role of the element within the documented procedure. To set the description right click on the element and choose the Open With > Description Editor menu item (See Figure 1.6 below). Enter the desired description text and save the changes (See Figure 1.7 below).



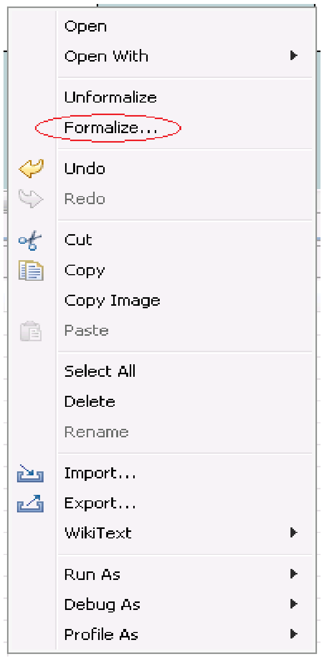


```
The pacer instance has an increased amount of signals del
Activity.  This allows the Pacer to deliver the appropriat
accomodate the heart under the increased activity.
```

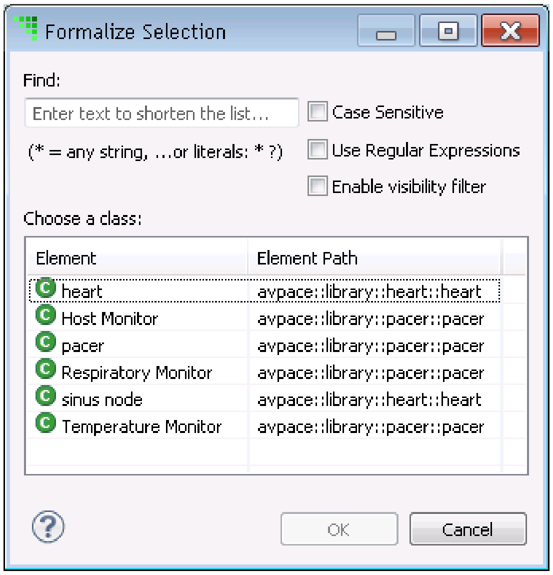
Figure 7

Formal participant creation:

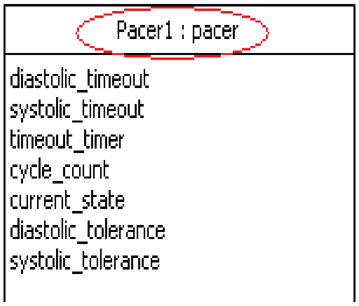
If there is a formal element present that the participant can represent, then right click on the participant and choose the Formalize… menu item (as shown in Figure 1.8). At this point there is no further configuration required. All data will be derived from the formal element chosen.

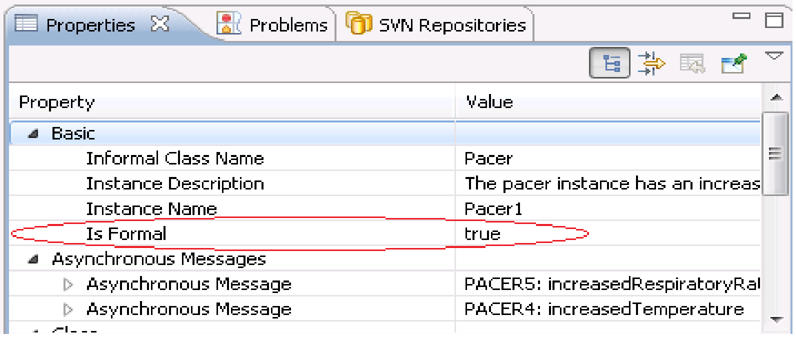


After executing the Formalize… menu item a wizard will appear. This wizard will present the available objects to formalize against. The elements will be listed in a flat list, giving the element name in the left column and the element's path in the right column (as shown in Figure 1.9). Locate the element of interest and click the OK button. The participant will now be formalized. This can be verified by inspecting the display name or in the Properties view while the element is selected (see Figure 1.10).


Formalize Selection
Find:
Enter text to shorten the list…
(* = any string, …or literals: * ?)
□ Case Sensitive
□ Use Regular Expressions
□ Enable visibility filter
Choose a class:
OK
Cancel

Element	Element Path
heart	avpace::library::heart::heart
Host Monitor	avpace::library::pacer::pacer
pacer	avpace::library::pacer::pacer
Respiratory Monitor	avpace::library::pacer::pacer
sinus node	avpace::library::heart::heart
Temperature Monitor	avpace::library::pacer::pacer



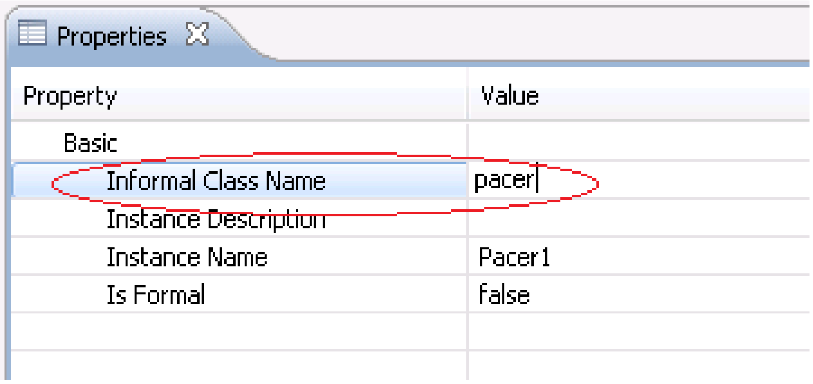


Property	Value
Basic	
Informal Class Name	Pacer
Instance Description	The pacer instance has an increas
Instance Name	Pacer1
Is Formal	true
Asynchronous Messages	
Asynchronous Message	PACER5: increasedRespiratoryRal
Asynchronous Message	PACER4: increasedTemperature

Figure 10

Informal participant creation:

If the element created is informal, then right click and the element and choose Rename. Enter the desired name of the element (note that it should be consistent with the future name of the realized element) in the dialog box that appears. If the element created is the Instance participant type further detail can be set. The Instance participant type can have an Instance name and an Informal Class name. The Informal Class name is used to depict the object type that the Instance is instantiated from. The Instance name is a unique name that represents this particular instance. Both of these are set in the Properties view. To set them select the Instance participant element and switch to the Properties view (See Figure 1.11). Find each attribute and change the right field to the appropriate values.



Property	Value
Basic	
Informal Class Name	pacer
Instance Description	
Instance Name	Pacer1
Is Formal	false

Figure 11

The Instance and Class participants can additionally have informal Attribute Values and Attributes added. This will aid in further detail for the procedure that is being captured. The informal Attribute Value object can additionally have a value set; this is a value for the attribute in the given procedure. To create an informal Attribute Value, choose the Instance participant of interest and right click. From the context menu choose the New > Attribute menu item (See Figure 1.12). The informal Attribute Value will be added to the Instance participant body. To rename the Attribute Value open the Model Explorer view and right click on the Attribute Value instance. In the dialog that appears set the desired name. As with other informal elements take care in naming the Attribute Value as at a later time it will become a real Class Attribute. To configure a value for the Attribute Value instance open the Model Explorer view and select the element in the tree. In the properties view navigate to the Attribute Value field and set the textual value. The informal Attributes can be added to the Class participant. Creation of the informal Attribute is identical to the steps above for the informal Attribute Value, only a Class participant is selected rather than an Instance participant. Renaming is also completed in the same steps as above for the informal Attribute Value. The informal Attribute can additionally have a type set. The type is a simple unchecked string. Carefully name the type as this will be a real type in a future model. To set the type open the Model Explorer view and select the informal Attribute. In the properties view locate the Informal Attribute Type field and set the appropriate value.

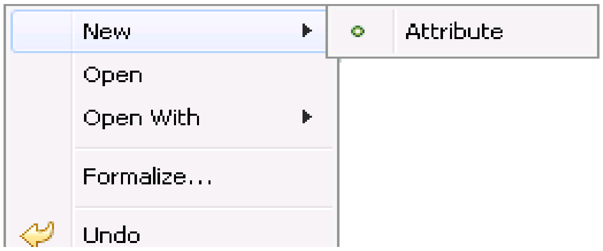


Figure 12

Link creation:

Again, as with other elements, the proper tool needs to be selected for Link creation. This tool is found in the Palette view under the Communication folder. Select this tool in the palette view and left click the target element. This will be one of the participants that are associated. Drag the mouse to the other target (on the other end of the association) and release the mouse button. As with the participant elements it is good practice to add a description for the Link. Do this in the same way as stated above. The description should clearly explain to readers the type of Link (association or interface satisfaction) and should give details on the realized association's configuration.

Additional data can be configured to further detail the Link. The available attributes are:

Association Number
End Text Phrase
End Visibility
Start Text Phrase
Start Visibility

The Association Number, End Text Phrase, and Start Text Phrase values are only necessary when dealing with informal elements and will be described in the section below titled Informal Link creation.

The End Visibility and Start Visibility attributes are used in both the formal and informal cases. The start of the Link is considered the end at the participant that the Link was drawn on first. The values of the two attributes can be changed in the Properties view while the Link is selected. Once set to visible an open arrow will be drawn at the corresponding end. These attributes are used to describe the possible navigation directions. In BridgePoint all associations are considered bi-directional, so for the formal case these attributes are ignored. This is something to consider when creating an informal Link that will at some point be realized.

Formal Link creation:

If the underlying participants have realized associations, then the Link should be formalized at this stage. Only Class Associations are supported for formalizing the Link. Only Links that are between to Instance participants may be formalized. To formalize a Link right click and choose the Formalize… menu item (See Figure 1.3). A dialog is opened that will present all possible associations between the formal Class objects. The association of interest shall be chosen using the dialog's pull down menu. Once the association is chosen click the Finish button. At this point the Link will be formal, and the various Association Name, Start and End Text Phrases will be derived.

Informal Link creation:

If creating an informal Link, the attributes mentioned above in the Link creation section must be set. How to set the Start and End Visibility attributes is described in the same section. The remaining attributes to set are Association Number, End Text Phrase and Start Text Phrase. These are also set in the Properties view when the Link is selected. For each enter an appropriate value in the right field for the given attribute. Take consideration of the values entered as the Link will become realized at some point.

Message creation:

Message creation is achieved in the same way as Link creation, in that the proper tool must be selected. There are three Message tools, Synchronous Message, Asynchronous Message and Return Message. The proper tool depends on the procedure that is being documented. Creation is done the same way for each. Once the proper tool is selected, click the left mouse button down at the desired start location. Drag the mouse to the desired end location and release the mouse button. Messages should be created near the Link for which they are delivered across. Messages do not connect to a start or end element. As with all other elements it is good practice to enter a description about the created Message. This description should describe the Message's role in the procedure being documented.

A Message can be further configured through the following attributes:

- Guard Condition
- Return Value
- Result Target
- Sequence Number

Each of these attributes is valid in both the formal and informal case. Formalizing a message does not populate these values; therefore in each case values should be set where it makes sense. The attributes are set in the Properties view when the Message is selected. For each Message set the appropriate value in the right field. Take consideration when setting the values in the formal case as the values are not checked against the realized data. For instance you can enter anything for the Return Value attribute, however it must match what is truly returned by the Message delivered.

The Sequence Number attribute can be used to add chronological information to the Message as it fits within the documented procedure.

Formal Message creation:

When creating a Message that will be formal, the target element must be chosen. The act of choosing is completed during the formalization step. To achieve this, the Message is first selected and then the target element is selected. This results in two elements being selected. After selecting the elements involved the Formalize… menu item should be executed (see Figure 1.3). A dialog is opened that lists the available Messages that exist in the target (the target is the one which owns the Message that is to be called). Click the Finish button once the appropriate Message is selected.

Informal Message creation:

In additional to configuring what is listed in the above Message creation section, name the Message. This is done by right clicking on the Message and selecting the Rename menu item. In the dialog that is shown enter a name for the Message. Take consideration in the value entered as the Message will become realized at some point.

Informal Messages may also have arguments added. This will aid in providing a clear document for the realized elements that must be completed. Creating a new argument is achieved by right clicking on the Message and selecting the New > Argument menu item. At this point the Argument should be renamed to something meaningful. Right click on the Argument in the Model Explorer view and choose the Rename menu item. On the dialog that appears enter a good name. The informal Argument can be further configured to include a value for the given procedure. This can be achieved by first selecting the Argument in the Model Explorer view, then by changing the Argument Value field in the Properties view. Note that the string is unchecked, therefore care needs to be taken so that when realized the types are compatible.

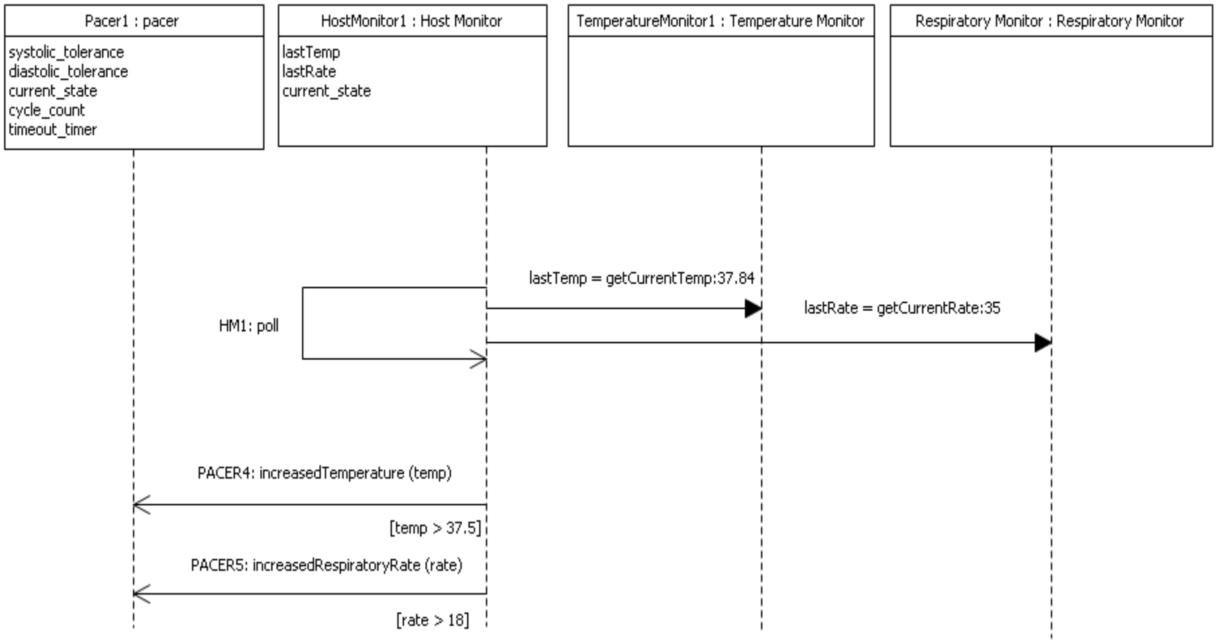
Recommendations

As with any programming language, UML models should be factored into the smallest amount of data that makes sense. It is easier to read and understand a diagram (or program code) when the subjects within the system are factored into many small functional parts. This prevents the diagram from growing large to a point where readability is hindered.

The Communication Diagram is no exception. As stated in the Purpose section above, the Communication Diagram is used to document a single procedure. The size of the procedure can vary, but care needs to be taken when documenting such that large procedures are broken in to smaller procedures. Objects in UML have many procedures that are performed throughout the life of the system. The Communication Diagram was not created to document them all in one diagram. Always take care to include only those objects, links and messages that are pertinent to the procedure being documented. Always include descriptions, for the elements, that describe their role in the documented procedure.

If a diagram is growing to the point where the picture is hard to follow, take a step back and consider the possibilities for refactoring. Readers can always be referred to a set of documents to help describe a larger procedure.

Sequence diagram


Pacer1 : pacer
systolic_tolerance
diastolic_tolerance
current_state
cycle_count
timeout_timer
HostMonitor1 : Host Monitor
lastTemp
lastRate
current_state
TemperatureMonitor1 : Temperature Monitor
Respiratory Monitor : Respiratory Monitor
lastTemp = getCurrentTemp:37.84
lastRate = getCurrentRate:35
HM1: poll
PACER4: increasedTemperature (temp)
[temp > 37.5]
PACER5: increasedRespiratoryRate (rate)
[rate > 18]

The Sequence Diagram shows communication between objects and the communications between them with a time perspective.

Use a Sequence Diagram to document the details for an interaction. If the structure of the objects included in the interaction is not fully known, consider creating a Communication diagram first. While Sequence and Communication diagrams detail much of the same information, they are complimentary. Each is good at detailing certain aspects. Sequence diagrams allow for defining finer details of the control, while Communication diagrams detail the structure of the procedure.

Things to Know

Participant (Component, Instance, Actor, External Entity, Class, Package)

Message (Synchronous, Asynchronous, Return)

Lifeline

Time Span

Timing Mark

When to use

When a clear picture of an interaction between a set of elements is required, including messaging sequence and timing information, use a Sequence Diagram.

Sequence Diagrams can be used formally or informally. In the formal case the various objects and messages will already exist. The Sequence elements will be formalized against the existing objects and will be used to depict the interaction that occurs among the existing objects. The resulting document will aid in detailing a single interaction, while filtering out all possible interactions among the objects.

In the informal case there will be no existing objects or messages. The Sequence elements will be created to document a required interaction. The elements can then be used to realize the objects and the messages required to execute the interaction. When creating an informal document a few things must be known. The following checklist can be used to get started:

1. The required interaction and its goal, what must be accomplished.
2. The various objects that participate in the interaction.
3. The messages between the participating objects that are called in order to achieve the interaction goal.

The resulting document is living and may change with time. When focusing on the informal case, the exact objects and messages do not have to be precise. The document will change as these elements come to realization.

A well written Sequence Diagram will be short and precise at describing the desired interaction. Only include objects that participate in the interaction, and only include messages between the objects that participate in the interaction. If the interaction to be documented is large, consider breaking the interaction into sub-interactions each with their own Sequence Diagram. When breaking up a large interaction into multiple interactions use a package hierarchy and naming scheme to allow easy navigation for the reader. An example follows:

Descriptive Interaction Name (Package)

|_ Descriptive Sub Interaction Name Step 1 (Package)

|_ Descriptive Sub Interaction Name Step 2 (Package)

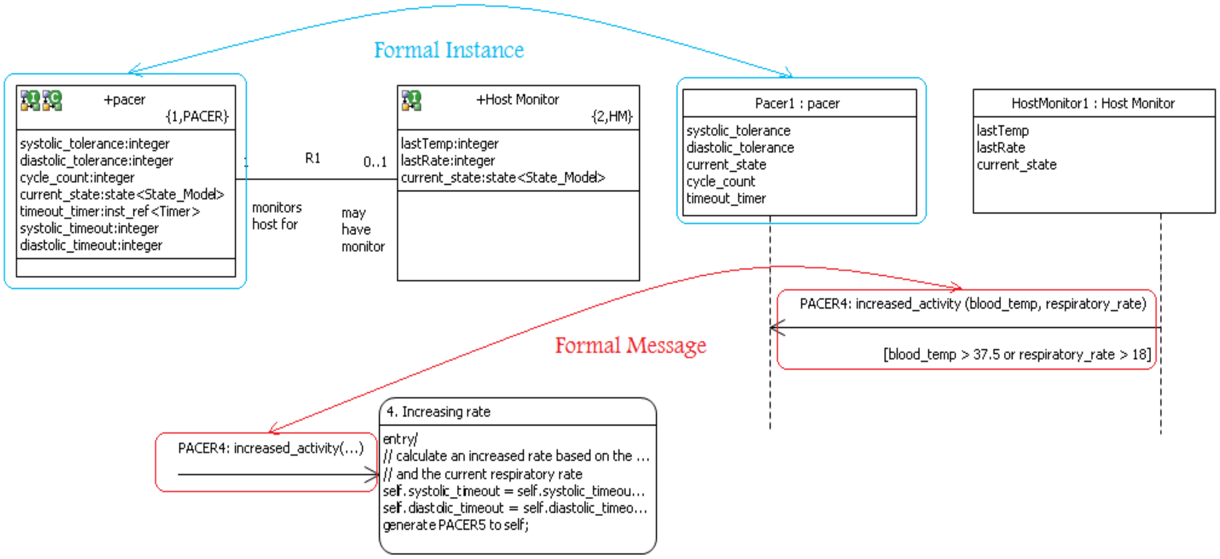

Formal Instance
+pacer
{1,PACER}
systolic_tolerance:integer
diastolic_tolerance:integer
cycle_count:integer
current_state:state<State_Model>
timeout_timer:inst_ref<Timer>
systolic_timeout:integer
diastolic_timeout:integer
1
R1
0..1
monitors host for
may have monitor
+Host Monitor
{2,HM}
lastTemp:integer
lastRate:integer
current_state:state<State_Model>
Pacer1 : pacer
systolic_tolerance
diastolic_tolerance
current_state
cycle_count
timeout_timer
HostMonitor1 : Host Monitor
lastTemp
lastRate
current_state
PACER4: increased_activity (blood_temp, respiratory_rate)
[blood_temp > 37.5 or respiratory_rate > 18]
Formal Message
PACER4: increased_activity(...)
4. Increasing rate
entry/
// calculate an increased rate based on the ...
// and the current respiratory rate
self.systolic_timeout = self.systolic_timeou...
self.diastolic_timeout = self.diastolic_timeo...
generate PACER5 to self;

Figure 1 Formal referencing

Usage

Meaning of Message source and target:

There is meaning in where messages are drawn from and to. Messages can be drawn from or to the symbol or a Communication Line (Lifespan). When a message is drawn from or to a symbol this indicates a static call, where the element does not need to be instantiated. When a message is drawn from or to a Communication Line this usually indicates an instance call where the target is instantiated. The BridgePoint tool allows a Communication Line on all elements, even those that are never instantiated. This allows for easier diagram formatting, as otherwise the message space allowed is limited by the target symbol's size. Take note that when a message is drawn to a Communication Line with these aspects, it does not represent an instance call. Use of the Communication Line in this aspect is optional; consistency shall be used in each individual organization.

Formal Usage:

As stated above in the formal case the objects and messages will already exist. A Package is used to enclose sequence elements and thus represents the Sequence Diagram. Create the Sequence Diagram at a location that makes sense given the existing elements.

Select the appropriate tool to create a participant. This will be one of the listed participants from the Elements section above. Choose the tool that will match the formal object. For instance if the content of the Sequence Diagram will show the interaction of Components, then choose the Component Participant tool. Create one participant for every object that will participate in the documented interaction. For each participant created, use the formalization wizard to locate and use the appropriate object. Create a Communication Line for all objects that are instantiated in the documented interaction or for those not instantiated where the diagram formatting approach is used.

Now create the necessary messages that are documented by the interaction. In most cases this should not be all of the possible messages that are between two associated objects. The diagram shall focus on the interaction to document and shall ignore any messages that are not pertinent to that interaction. Create the messages between the caller and target objects. As detailed in the "Meaning of Message source and target" section, messages can be drawn from and to multiple locations. For each message use the formalization wizard to locate and use the appropriate action.

Informal Usage:

When creating the Sequence Diagram informally, there are no existing elements to base the interaction on. The diagram will serve to provide aid when creating the real elements. An informal diagram will start off informal but will later migrate to a formal diagram. While creating the diagram consider carefully the interaction to be documented, taking care to name the informal elements appropriately.

The mechanics of creating this diagram are identical to that of the formal case. The one step that will not be taken is using the formalization wizard.

The informal Sequence Diagram elements have attributes that can be set which allow further definition. Informal elements can be configured such that enough data is present to allow fully configured realized elements. The proceeding How To section will explain how these informal attributes are used.

Conversion to formal document:

As stated above at some point all Sequence Diagrams shall become formal. After an informal diagram has been agreed upon by all stake holders, it is time to realize the interaction. Create an object for each participant in the interaction. The object created depends on the participant type. The exact steps for such creation are not within the scope of this document. For example, if the participant type is component, then a matching component shall be created in a package.

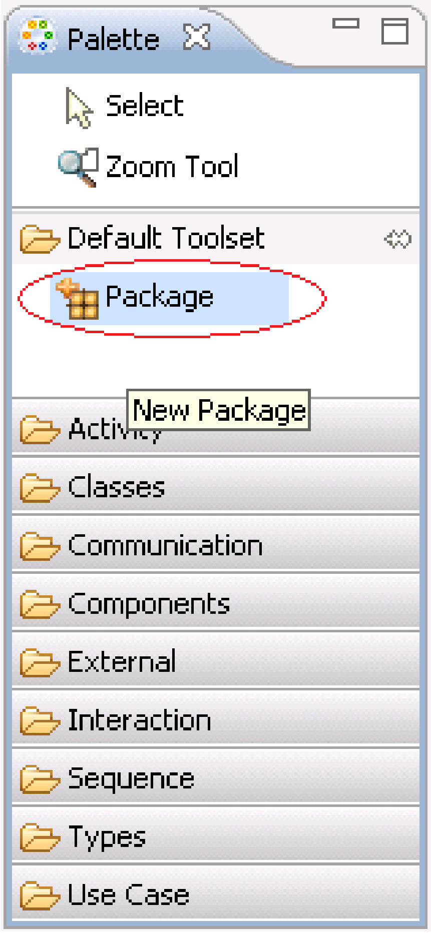
Create the necessary action, i.e., operation, for each message that exists in the Sequence Diagram. The destination for the actions will be the object that the participant represents at the end of the message on the diagram. The end of the message is denoted by the arrow end. Depending on where the message is drawn to (symbol or Communication Line) the created action may need to be specified as static (class-based for class operations).

After all of the objects and actions have been created it is time to formalize the original document. Select each participant and use the formalization wizard to locate and use the objects created above. Repeat this step for each message within the document.

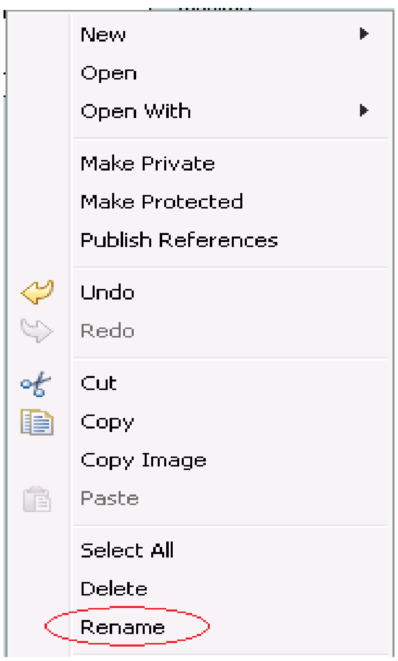
Steps to creating Informative Sequence Diagrams

Diagram creation:

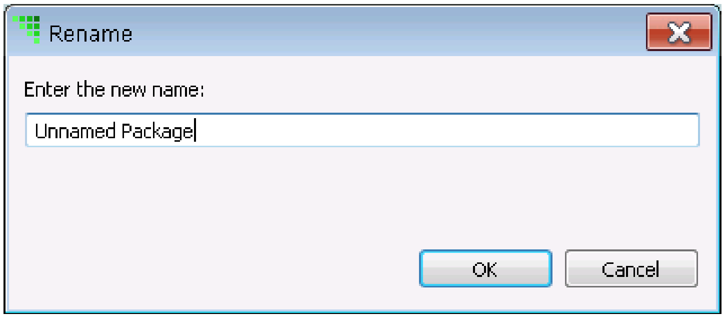
Once the location has been determined, locate the palette view (see Figure 1.2 below). In the palette view select the Package tool (as shown in Figure 1.2). On the editor page click the left mouse button and drag a distance. A marquee will be drawn that indicates the location and size of the graphical symbol to be created. Once the symbol is at a desired location and size let go of the left mouse button. At this point a new Package has been created. Right click on the Package graphical symbol and select "Rename" (See Figure 1.3 and 1.4 below). In the window that is opened enter a good name for the package. This name should reflect the interaction that is being captured by the Sequence Diagram.



Figure



Figure


Rename

Enter the new name:

Unnamed Package

Figure

Participation creation:

As with diagram creation, the proper tool must be selected. As stated in the Elements section there are many participants. Select the participant tool that is of interest (see Figure 1.5 below). Once selected proceed to draw the symbol for the participant graphical element (see steps in package creation for details). At this point the newly created participant can either be formalized or customized to represent a future formal element. For either case, formal or informal, it is a good idea to enter a description for the element created. The description shall describe the role of the element within the documented interaction. To set the description right click on the element and choose the Open With > Description Editor menu item (See Figure 1.6 below). Enter the desired description text and save the changes (See Figure 1.7 below).

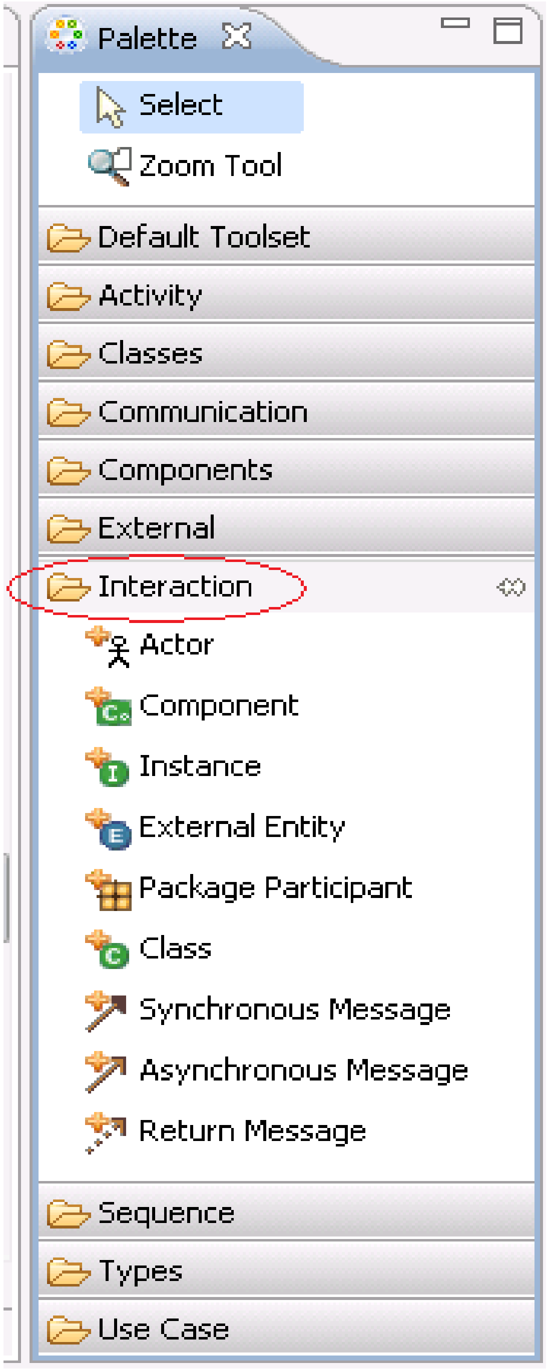
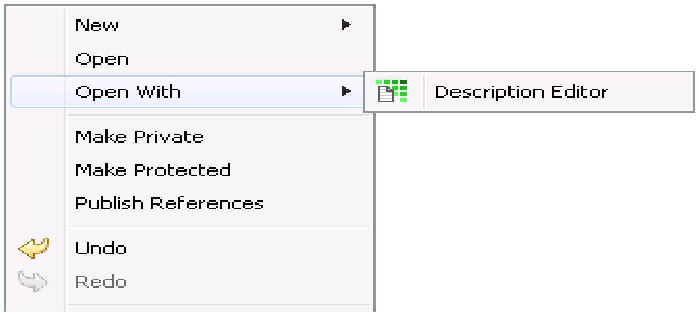
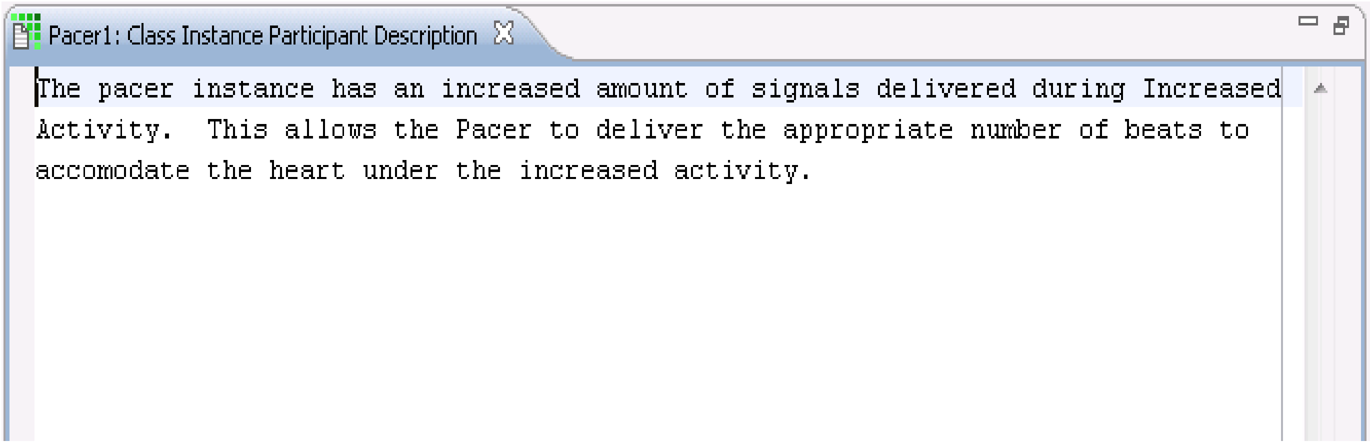

Palette
Select
Zoom Tool
Default Toolset
Activity
Classes
Communication
Components
External
Interaction
Actor
Component
Instance
External Entity
Package Participant
Class
Synchronous Message
Asynchronous Message
Return Message
Sequence
Types
Use Case

Figure 1.5



Figure

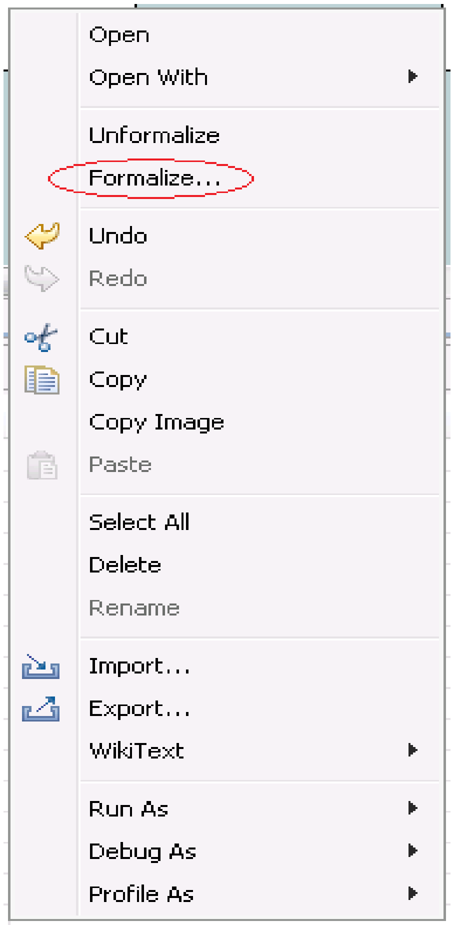


The pacer instance has an increased amount of signals delivered during Increased Activity. This allows the Pacer to deliver the appropriate number of beats to accomodate the heart under the increased activity.

Figure

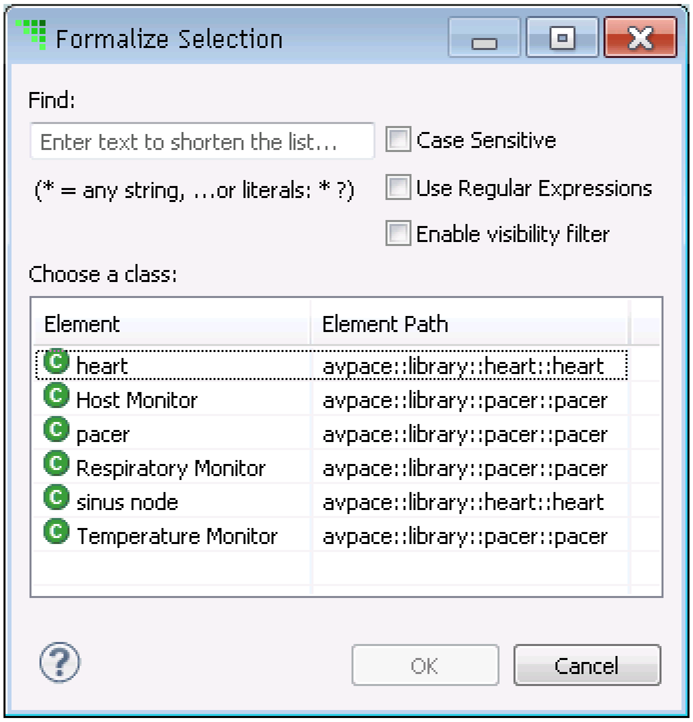
Formal participant creation:

If there is a formal element present that the participant can represent, then right click on the participant and choose the Formalize… menu item (as shown in Figure 1.8). At this point there is no further configuration required. All data will be derived from the formal element chosen.



Figure

After executing the Formalize… menu item a wizard will appear. This wizard will present the available objects to formalize against. The elements will be listed in a flat list, giving the element name in the left column and the element's path in the right column (as shown in Figure 1.9). Locate the element of interest and click the OK button. The participant will now be formalized. This can be verified by inspecting the display name or in the Properties view while the element is selected (see Figure 1.10).



Formalize Selection

Find:

Enter text to shorten the list... □ Case Sensitive

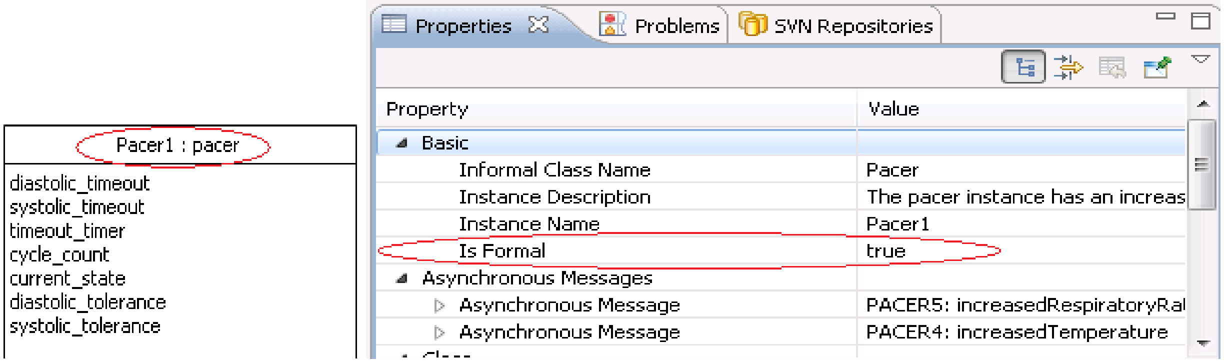
(* = any string, ...or literals: * ?) □ Use Regular Expressions

□ Enable visibility filter

Choose a class:

Element	Element Path
heart	avpace::library::heart::heart
Host Monitor	avpace::library::pacer::pacer
pacer	avpace::library::pacer::pacer
Respiratory Monitor	avpace::library::pacer::pacer
sinus node	avpace::library::heart::heart
Temperature Monitor	avpace::library::pacer::pacer

Figure



Pacer1 : pacer

diastolic_timeout
systolic_timeout
timeout_timer
cycle_count
current_state
diastolic_tolerance
systolic_tolerance

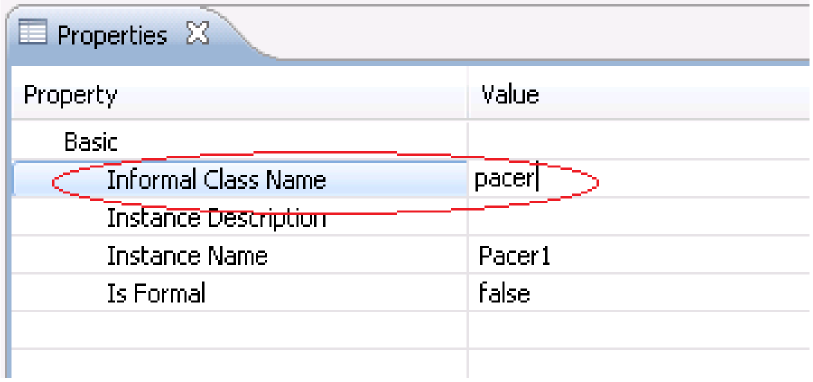
Property	Value
Basic	
Informal Class Name	Pacer
Instance Description	The pacer instance has an increas
Instance Name	Pacer1
Is Formal	true
Asynchronous Messages	
Asynchronous Message	PACER5: increasedRespiratoryRa
Asynchronous Message	PACER4: increasedTemperature

Figure

Informal participant creation:

If the element created is informal, then right click on the element and choose Rename. Enter the desired name of the element (note that it should be consistent with the future name of the realized element) in the dialog box that appears. If the element created is the Instance participant type further detail can be set. The Instance participant type can have an Instance name and an Informal Class name. The Informal Class name is used to depict the object type that the Instance is instantiated from. The Instance name is a unique name that represents this particular instance.

Both of these are set in the Properties view. To set them select the Instance participant element and switch to the Properties view (See Figure 1.11). Find each attribute and change the right field to the appropriate values.



Property	Value
Basic	
Informal Class Name	pacer
Instance Description	
Instance Name	Pacer1
Is Formal	false

Figure 1

The Instance and Class participants can additionally have informal Attribute Values and Attributes added. This will aid in further detail for the interaction that is being captured. The informal Attribute Value object can additionally have a value set; this is a value for the attribute in the given interaction. To create an informal Attribute Value, choose the Instance participant of interest and right click. From the context menu choose the New > Attribute menu item (See Figure 1.12). The informal Attribute Value will be added to the Instance participant body. To rename the Attribute Value open the Model Explorer view and right click on the Attribute Value instance. In the dialog that appears set the desired name. As with other informal elements take care in naming the Attribute Value as at a later time it will become a real Class Attribute. To configure a value for the Attribute Value instance open the Model Explorer view and select the element in the tree. In the properties view navigate to the Attribute Value field and set the textual value. The informal Attributes can be added to the Class participant. Creation of the informal Attribute is identical to the steps above for the informal Attribute Value, only a Class participant is selected rather than an Instance participant. Renaming is also completed in the same steps as above for the informal Attribute Value. The informal Attribute can additionally have a type set. The type is a simple unchecked string. Carefully name the type as this will be a real type in a future model. To set the type open the Model Explorer view and select the informal Attribute. In the properties view locate the Informal Attribute Type field and set the appropriate value.

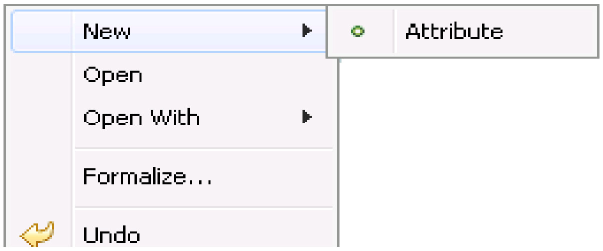
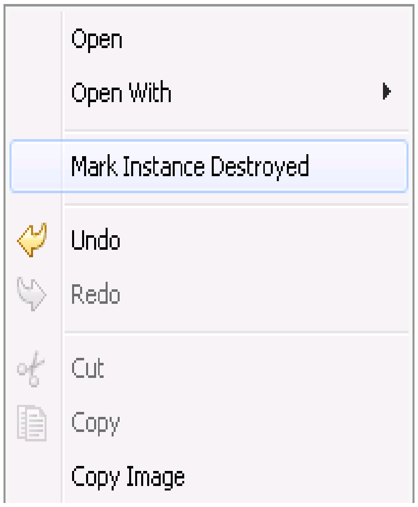


Figure 1

Communication Line creation:

Communication Lines can be added to any symbol. See the “Meaning of Message source and target” section above for additional detail about the Communication Line. To create a Communication Line select the proper tool under the Sequence tool folder in the palette view. Once the proper tool is selected, click the left mouse button down at the desired start location. Drag the mouse to the desired end location and release the mouse button. Sequence diagrams depict time vertically, meaning that messages near the top of the document are considered to be sent before lower ones. Given this the Communication Line should be drawn vertically. A Communication Line can be terminated (See Figure 1.13 below). This indicates that the Instance has been destroyed. To configure this select the Communication Line and right click. Choose the Mark Instance Destroyed menu item (See Figure 1.14 below).

Figure 1.13



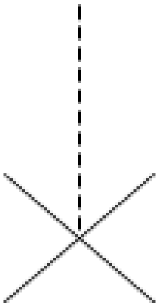


Figure 1.14 (Destroyed Instance)

Message creation:

Message creation is achieved in the same way as Communication Line creation, in that the proper tool must be selected. There are three Message tools, Synchronous Message, Asynchronous Message and Return Message. The proper tool depends on the interaction that is being documented. Creation is done the same way for each. Once the proper tool is selected, click the left mouse button down at the desired start location. Drag the mouse to the desired end location and release the mouse button. See the "Meaning of Message source and target" section above for details about where messages are drawn from and to. As with all other elements it is good practice to enter a description about the created Message. This description should describe the Message's role in the interaction being documented.

A Message can be further configured through the following attributes:

Guard Condition
Result Target
Return Value
Sequence Number

Each of these attributes is valid in both the formal and informal case. Formalizing a message does not populate these values; therefore in each case values should be set where it makes sense. The attributes are set in the Properties view when the Message is selected. For each Message set the appropriate value in the right field. Take consideration when setting the values in the formal case as the values are not checked against the realized data. For instance you can enter anything for the Return Value attribute, however it must match what is truly returned by the Message delivered.

The Sequence Number attribute can be used to add chronological information to the Message as it fits within the documented interaction.

Formal Message creation:

To formalize a message select it in the diagram. After selecting the message the Formalize… menu item should be executed (see Figure 1.3). A dialog is opened that lists the available Messages that exist in the target (the target is the one which owns the Message that is to be called). Click the Finish button once the appropriate Message is selected.

Informal Message creation:

In additional to configuring what is listed in the above Message creation section, name the Message. This is done by right clicking on the Message and selecting the Rename menu item. In the dialog that is shown enter a name for the Message. Take consideration in the value entered as the Message will become realized at some point.

Informal Messages may also have arguments added. This will aid in providing a clear document for the realized elements that must be completed. Creating a new argument is achieved by right clicking on the Message and selecting the New > Argument menu item. At this point the Argument should be renamed to something meaningful. Right click on the Argument in the Model Explorer view and choose the Rename menu item. On the dialog that appears enter a good name. The informal Argument can be further configured to include a value for the given interaction. This can be achieved by first selecting the Argument in the Model Explorer view, then by changing the Argument Value field in the Properties view. Note that the string is unchecked, therefore care needs to be taken so that when realized the types are compatible.

Timing Mark and Span creation:

Additional timing information can be captured on the Sequence diagram. The amount of time that is spent between messages is captured by using Timing Marks along with Time Spans. These can be helpful when timing constraints need to be documented. These are created in the same way as the Communication Line, where the tools are found under the Sequence tool folder.

Once the tool is selected left click on the starting point along the Communication Line, then drag and let go of the mouse at the desired end location (this will not be connected to symbol). Each piece can have a label, which is free text. The Time Span label is enclosed with curly braces automatically. The labels are useful in adding detail about the specific occurrence at that point in time. To set the labels select the element of interest and right click. On the menu that appears select the "Rename" menu item (See Figure 1.2 and Figure 1.3). On the dialog that appears enter the desired label. For an example of Timing Mark and Time Span usage see Figure 1.15.

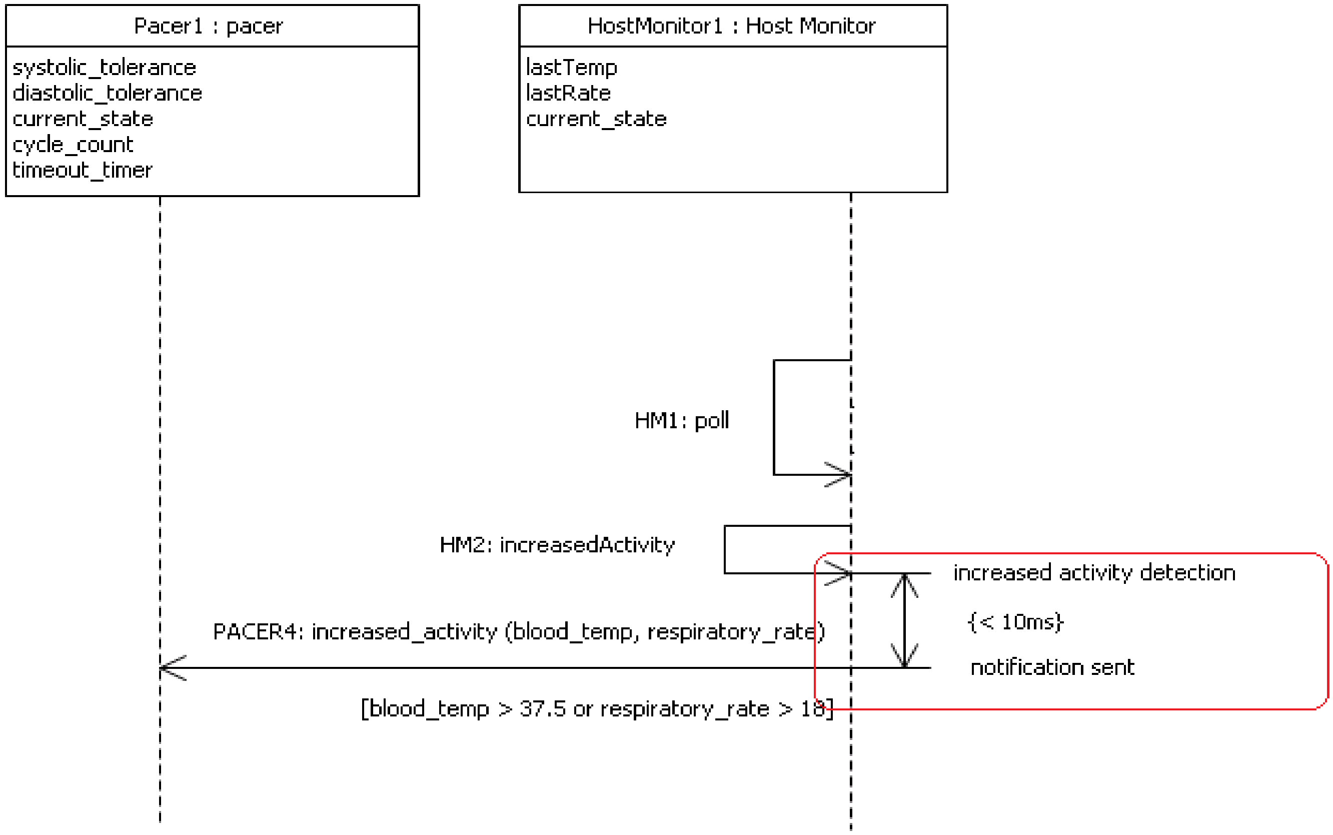


Figure 1.15

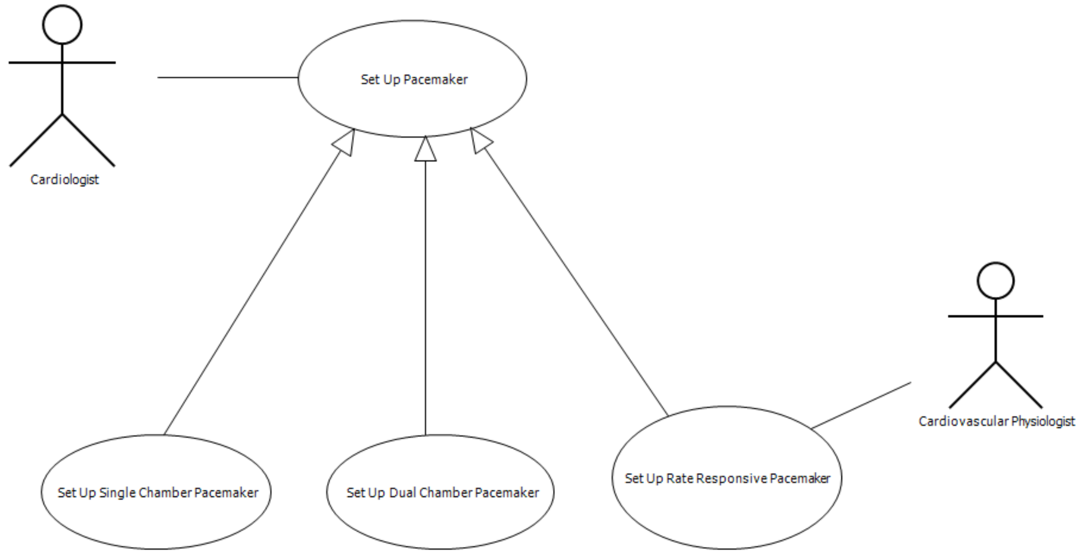
Recommendations

As with any programming language, UML models should be factored into the smallest amount of data that makes sense. It is easier to read and understand a diagram (or program code) when the subjects within the system are factored into many small functional parts. This prevents the diagram from growing large to a point where readability is hindered.

The Sequence Diagram is no exception. The size of an interaction can vary, but care needs to be taken when documenting such that large interactions are broken in to smaller sub-interactions. Always take care to include only those objects and messages that are pertinent to the procedure being documented. Always include descriptions, for the elements, that describe their role in the documented interaction.

If a diagram is growing to the point where the picture is hard to follow, take a step back and consider the possibilities for refactoring. Readers can always be referred to a set of documents to help describe a larger interaction.

Use case diagram


Set Up Pacemaker
Cardiologist
Cardiovascular Physiologist
Set Up Single Chamber Pacemaker
Set Up Dual Chamber Pacemaker
Set Up Rate Responsive Pacemaker

Use Cases are used to help organize the captured requirements, and help establish the subject matters for which we will build executable models. Use Case diagrams depict the high level goals of the system as Use Cases and the external agents that initiate them as Actors. Actors which use particular Use Cases are tied together by the 'uses' relation. Use Cases and Actors are grouped by similarity, using the 'generalization' relation. Use Cases may be referred to more than once using the 'includes' and 'extends' relations.

Things to Know

Actor

Use Case

Relationship (generalize, uses, includes, extends)

When to Use

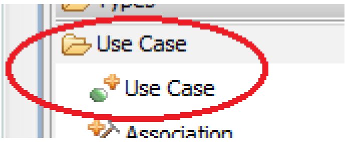
Creating Use Case Diagrams is typically the first task after organizing the requirements into the repository. These diagrams are not executable, but instead help identify the subject matters that will be explored and analyzed later to produce executable models. Because the diagrams are informal, they are useful for facilitating feedback with the requirements originators.

Creation

Create or open a Package to contain the Use Case under construction

Use Case

Locate the tool palette and open the Use Case section. In this section, locate the Use Case tool and select it.

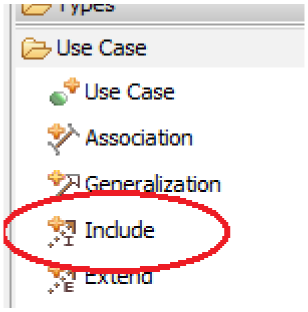


Move to the diagram and choose a suitable space on the sheet. Click on the left mouse button and drag the mouse until the outline is the size you want and release. The Use Case symbol will be drawn.

Now click the right mouse button on the Use Case symbol and select Rename... and provide the selected name. Double click the Use Case to open it's description editor and type in the two or three sentence description captured for it and save by clicking on the disk icon in the top menu bar, choosing Save in the File menu, or by typing <Ctrl+S>.

Include

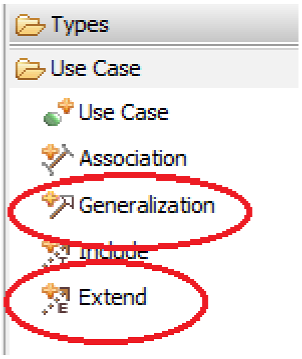
In the Use Case section of the tool palette, locate the Include tool and select it.



Click the left mouse button over the Use Case you wish to include another and drag the mouse over to the Use Case that you want to be included in the first. Release the mouse button and the include relation is drawn. You do not need to accurately click on the edges of a Use Case, drag from center to center and the tool will automatically take care of drawing the lines neatly. Right click on the new connector and choose Open. The include description editor opens. Type in the two or three sentence description found during the Use Case creation process and save it.

Extend and Generalize

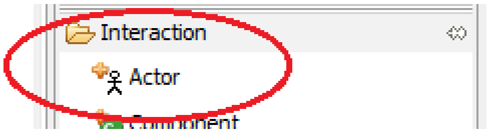
Locate the tools for these in the Use Case section of the tool palette.



They are drawn in exactly the same way as the Include relation. Generalization relations may also be drawn between Actors. Generalizations drawn between Actors and Use Cases are not allowed.

Actor

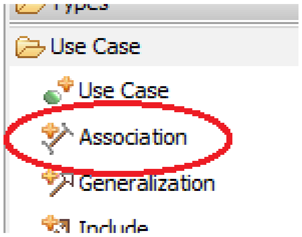
Locate the tool palette and open the Interaction section. The Actor symbol is used on a number of other diagrams and so is located in this more general section. You can keep both the Use Case and Interaction sections of the tool palette open at the same time.



Locate the Actor tool and select it. Move to the diagram and choose a suitable space on the sheet. Click on the left mouse button and drag the mouse until the outline is the size you want and release. The Actor symbol will be drawn. Click the right mouse button over the new Actor and select Rename... Name the Actor based on the name allocated during the Use Case creation procedure. Double click to open the Actor's description editor. Enter and save the two or three sentence description found during the Use Case creation procedure.

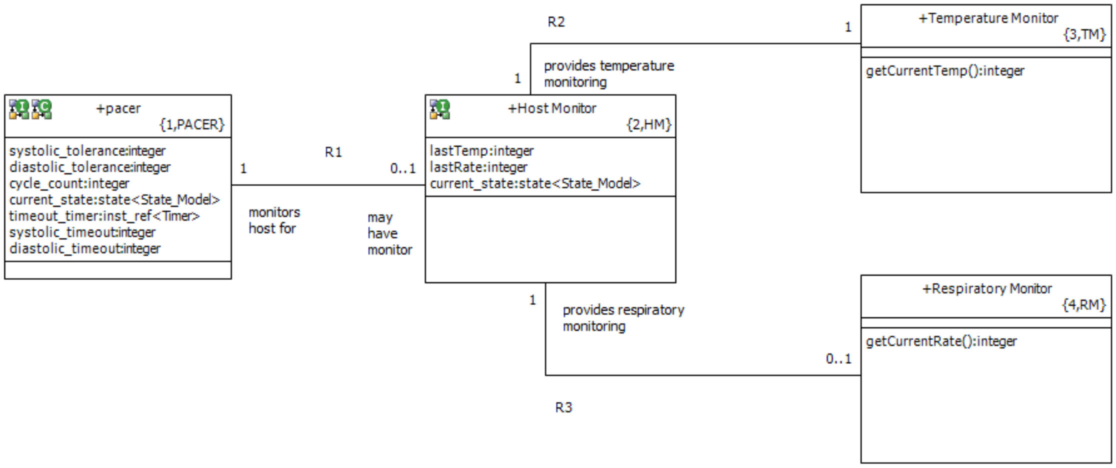
Uses (Association)

Locate the tool palette and open the Use Case section. Locate the Association tool and select it.



Move to the diagram and choose the Actor symbol to start from. Click the left mouse button and drag the mouse until it is over the Use Case you wish to associate with the Actor. Release the mouse button and the Association will be drawn. Double click to open the Association’s description editor. Enter and save the two or three sentence description found during the Use Case creation procedure.

Class Diagram



A Class Diagram formalizes the knowledge discovered in the requirements and when complete presents an abstract solution to meeting the requirement. Modelers use Class Diagrams to abstract the requirements captured and organized in the Use Cases. A Class captures a set of characteristics and behaviors that are abstracted by studying the detailed interactions described by the Use Cases. Classes which exhibit similar characteristics or behavior are grouped using the Generalization association. Classes which have observable enumerable relationships with each other are connected with Associations that document and describe the constraints on the relationships.

Things to Know

Class -	Attribute, Operation
Association -	Multiplicity, Conditionality
Generalization -	Supertype, Subtype

When to Use

Class Diagrams may be used both formally and informally. That is, they may be created with all the necessary detail to allow execution, or they may be used to clarify or explore and document some aspect of the requirement without providing too much detail.

Class diagram construction requires two principal inputs; Component diagrams and Use Case diagrams. It is possible to create informal Class diagrams without a Component structure, but formal, executable models must be specified within the boundaries of a Component.

Creation

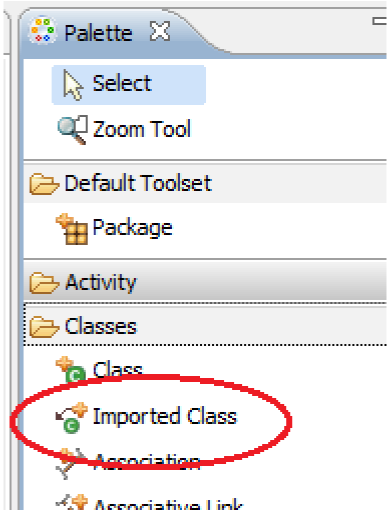
At a high level, there are two procedures for adding Classes to a diagram:

- Referring to a Class that is already modeled in another Package.
- Adding a new Class to the Package being worked on.

Before proceeding, ensure that you have the correct Package diagram open.

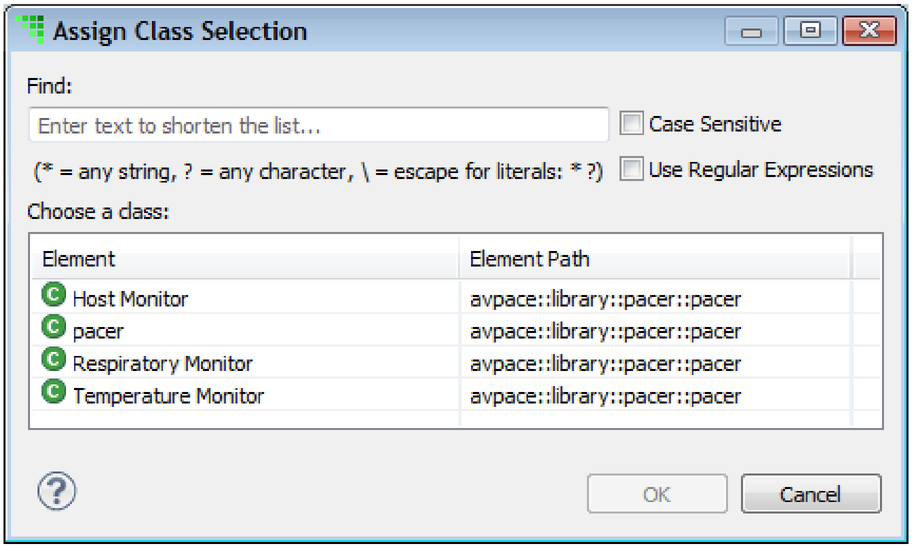
Imported Class

For Classes already defined elsewhere, locate the tool palette and open the Classes section. In this section locate the Imported Class tool and select it.



Move to the diagram and choose a suitable space on the sheet. Click on the left mouse button and drag the mouse until the outline is the size you want and release. The Imported Class symbol will be drawn.

Now click on the right mouse button and select Assign Class...



Assign Class Selection

Find:

Enter text to shorten the list...

□ Case Sensitive

(* = any string, ? = any character, \ = escape for literals: * ?)

□ Use Regular Expressions

Choose a class:

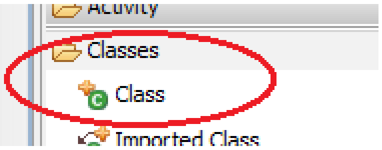
Element	Element Path
Host Monitor	avpace::library::pacer::pacer
pacer	avpace::library::pacer::pacer
Respiratory Monitor	avpace::library::pacer::pacer
Temperature Monitor	avpace::library::pacer::pacer

A Class Chooser dialog will appear showing the classes that you can choose from. If the list is very long, use the Find: field to narrow down your search.

When you have located the class you want to appear on the diagram, select it and click on OK, or just double click on the required class. The chooser dialog will disappear and the class will be shown on the diagram. There is no need for additional action; the Class already shows any Attributes and Operations already defined for it.

New Class

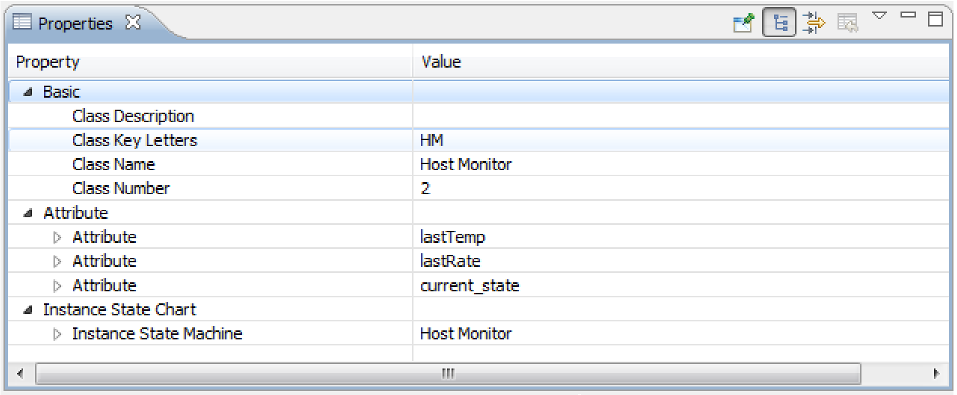
The procedure for adding a new Class to the Package is similar to that for the Imported Class, but using a different drawing tool. In the Classes section of the tool palette, locate the Class tool and select it.



Locate a suitable place in the diagram, click the left mouse button and drag until the rectangle is the size you want for your class. Release the mouse button and the Class symbol will be drawn.

Give the new Class a name by clicking the right mouse button and choosing Rename. Alternatively, open the Properties View to see and modify information captured about the

Class. If the Properties View is not already open, use Window > Show View > Other... > General > Properties to show it.



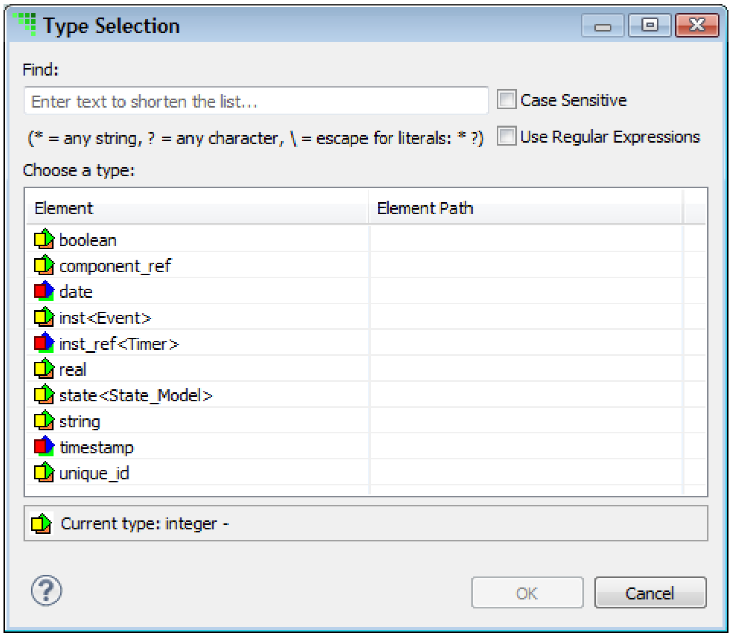
Property	Value
Basic	
Class Description	
Class Key Letters	HM
Class Name	Host Monitor
Class Number	2
Attribute	
Attribute	lastTemp
Attribute	lastRate
Attribute	current_state
Instance State Chart	
Instance State Machine	Host Monitor

Selecting any element on the diagram will update the Properties View.

Each Class is given a Key Letter as a short way to refer to it. Choose an appropriate short name and set it using the Properties View.

A new Class will need to have its Attributes added. Click the right mouse button over the Class and choose New > Attribute. Locate the Attribute in the Model Explorer View, right click on it and choose Rename. Give the new Attribute a name.

The new Attribute is created as an integer by default. If this is not what is required, click the right mouse button once more and choose Set Type...



Type Selection

Find:

Enter text to shorten the list...

☐ Case Sensitive

(* = any string, ? = any character, \ = escape for literals: * ?)

☐ Use Regular Expressions

Choose a type:

Element	Element Path
boolean	
component_ref	
date	
inst<Event>	
inst_ref<Timer>	
real	
state<State_Model>	
string	
timestamp	
unique_id	

Current type: integer -

A Data Type chooser dialog appears showing the Data Types available. Select or double click on the required Data Type to set the Attribute type.

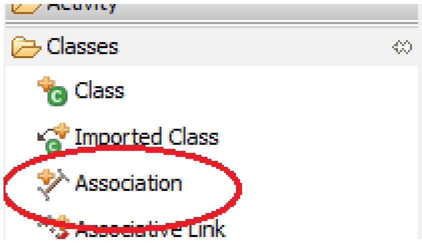
Associating Classes

There are two main kinds of relationship between Classes:

- Supertype/Subtype and
- Regular Association relationships.

New Association

In the Classes tool palette, choose the Association tool.

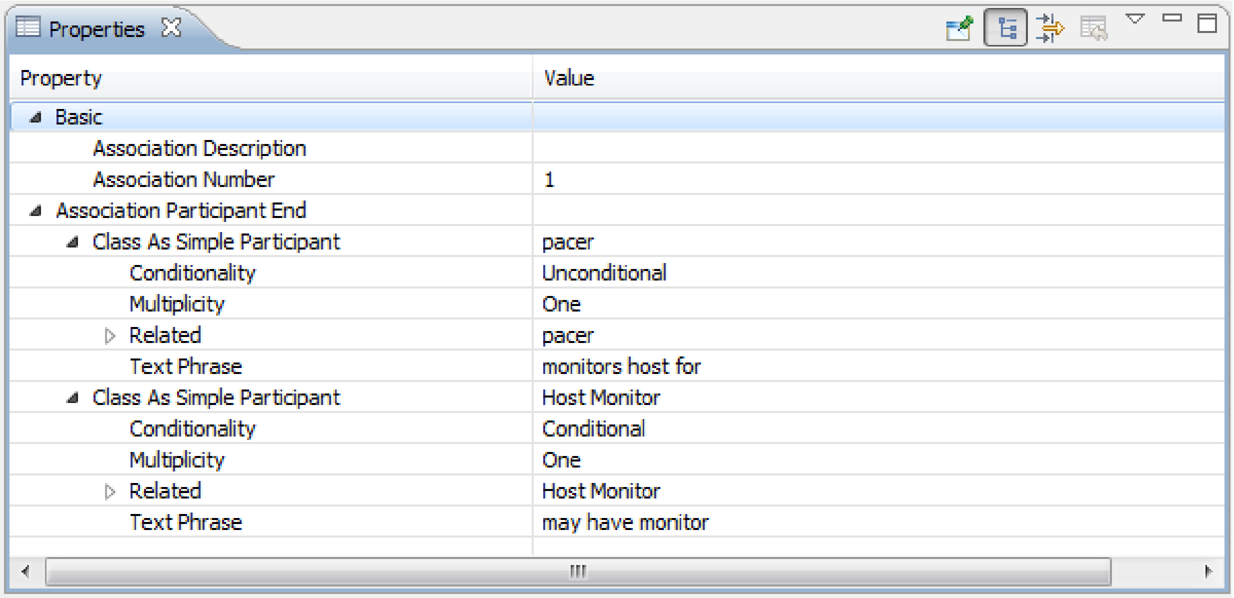


Move your mouse over one of the classes to be related. Click the left Mouse button and drag until the mouse is over the other class to be related. Now release the mouse button. The

Association will be created. Note that in most cases, you do not have to accurately click on the edges of a symbol, drag from center to center and the tool will automatically take care of drawing the lines neatly. You do need to take care of making the lines square on the diagram if that is what is desired.

Sometimes, you will want to show a reflexive Association, that is, one which returns to the same Class as it left. To draw this make sure you begin the drawing action near the center of the Class. Now drag the mouse a little without leaving the boundary of the Class. The tool will detect what you want to do and draw the looped back Association symbol for you.

With the Properties View showing (Window > Show View > Other... > General > Properties), click the left mouse button over the Association.



Property	Value
◢ Basic	
Association Description	
Association Number	1
◢ Association Participant End	
◢ Class As Simple Participant	pacer
Conditionality	Unconditional
Multiplicity	One
▷ Related	pacer
Text Phrase	monitors host for
◢ Class As Simple Participant	Host Monitor
Conditionality	Conditional
Multiplicity	One
▷ Related	Host Monitor
Text Phrase	may have monitor

Properties show the information you can set for the Association. Choose a number for the Association. This number is used in OAL to specify the Association to work on, so it is important that it is unique.

Setting Association Cardinality

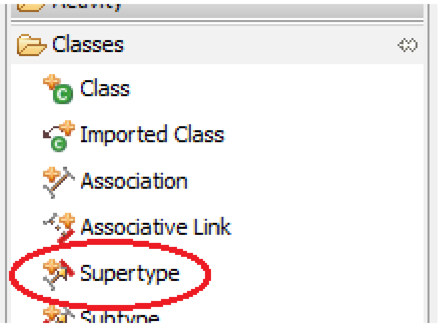
We need to specify the cardinality at each end of the association. The Properties View shows an Association Participant End section. Open it, and you see the Classes participating in the Association. Here, we can specify the cardinality information for each end of the Association. Open each participating Class name and set the Text Phrase. Now set the Conditionality and Multiplicity fields, using the following table to help you:

	Cardinality	Conditionality	Multiplicity
	1	Unconditional	One
	0..1	Conditional	One
	1..*	Unconditional	Many
	*	Conditional	Many

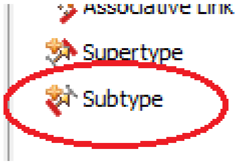
The diagram will update the Association symbol as you work. Confirm it looks the way you expected. You may use the mouse to drag the Association number and text phrases around to maximize readability. The cardinality symbols are not movable.

New Supertype/Subtype

Because there is usually just one supertype Class to many subtypes, drawing one of these is accomplished in two steps. First, locate the class that is to be the supertype. From the Classes tool palette, choose the Supertype tool.

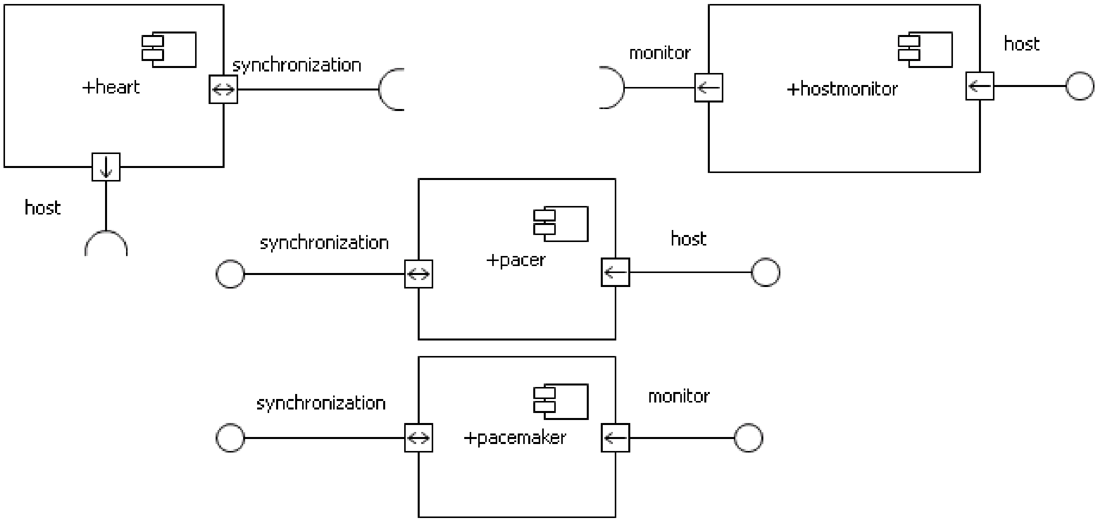


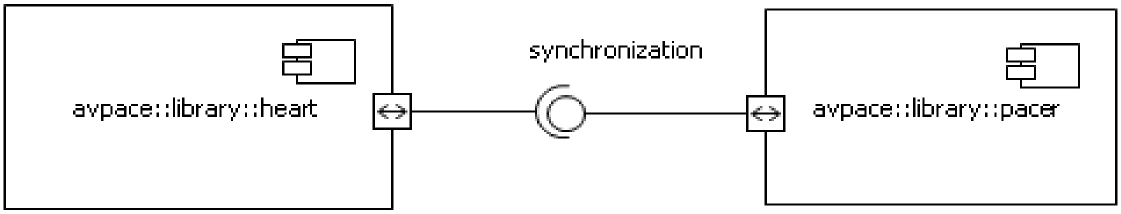
Click the left mouse button on the supertype Class and drag to place the supertype portion of the symbol where it is required. Now locate all the subtype Classes and for each one, select the Supertype tool and drag the connection from the subtype Class to the end of supertype portion of the symbol.

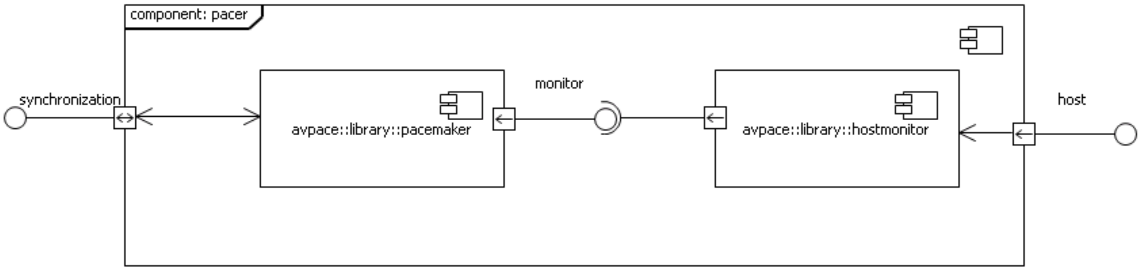


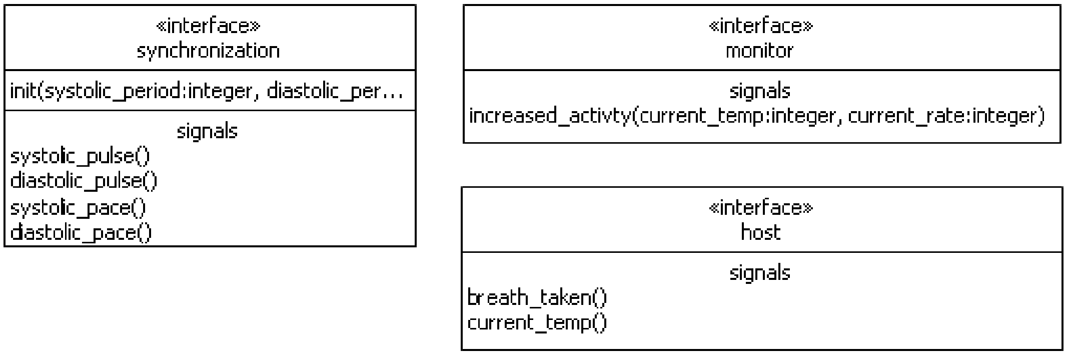
Absolute accuracy is not needed, so long as BridgePoint can tell which supertype symbol you meant, the tool will make the lines join neatly. Due to the nature of the Supertype/Subtype association, there is no cardinality or text phrase information required.

Component Diagram









Things to Know

Component	Component Reference
Provided Interface	Required Interface
Delegation	Interface
Interface Operation	Interface Signal
Parameter	

Definition

The Component Diagram captures the definition and internal structure for a Component of a system. It defines the various parts of a system and their communication channels. Modelers also use a Component Diagram to define the high level elements of a system. For each component, the internal structural view of the Component Diagram is used to define the executable behavior of each Component.

When to use

There are two types of Component Diagram, definition and internal structure. The definition type can also be broken into two distinct types, definition and reference.

The definition type is used to design the high level Components that make up a system. These are created first and define what will exist in a system and how each part will communicate.

The second flavor of the definition type is used to create contracts between the various Components. The reference name comes from the fact that the contracts are made between references of the definition Components. This allows for a solid definition of required Components, yet a multi-use reconfigurable usage of the Components. An example of this is a test bench. A test bench would replace one side of the contract with a custom version of the Component. The test bench Component can then verify the data that is sent from and to the concrete Component definition being tested.

Contracts are defined by Interface use. Interfaces are defined separately of the usage, and define the various signals and operations that are either provided or required when used. The contract is created when two interface usages are connected. A contract is made of one required interface (required signals/operations) and one provided interface (provided signals/operations). Contracts are made in the reference definition type.

The internal structure type allows for defining semantic details for a Component definition. This is used to define how the communication, when a contract is made, is handled. It allows for definition of both incoming and outgoing communication. The internal structure can be defined using internal Components, where the communication that is passed into or out of the Component is delegated to the internal Component. The internal structure can also be defined using classes, where incoming signals can be mapped to Class State Machine transitions. This document does not go into detail about this mapping. All internal elements of the Component have access to sending external data through the various signals and operations.

Things to know

1. The Components that make up the system under development.
2. The communication that may occur between each Component of the system.

Steps to creating a Component Diagram

Definition diagram creation:
Once the location has been determined, locate the palette view (see Figure 1.2 below). In the palette view select the Package tool (as shown in Figure 1.2). On the editor page click the left mouse button and drag a distance. A marquee will be drawn that indicates the location and size of the graphical symbol to be created. Once the symbol is at a desired location and size let go of the left mouse button. At this point a new Package has been created. Right click on the Package graphical symbol and select “Rename” (See Figure 1.3 and 1.4 below). In the window that is opened enter a good name for the package. This name should reflect the system that the Components are defined for.

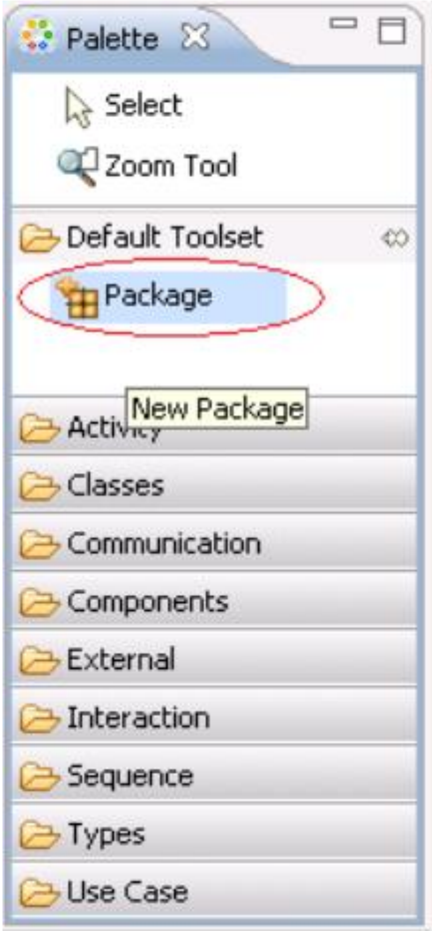


Figure 2

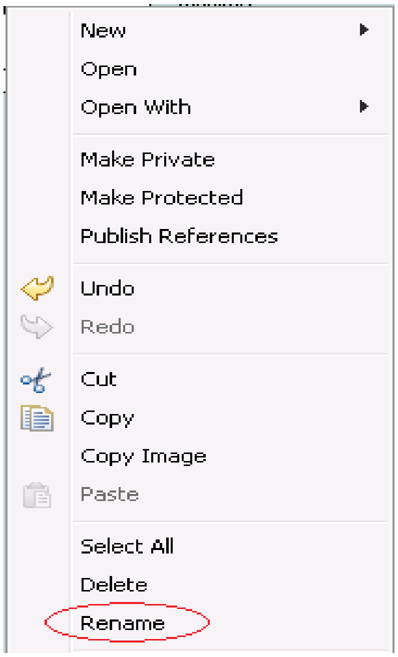
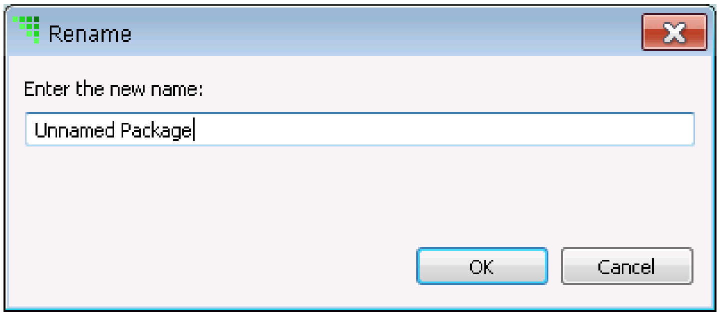


Figure 3



Enter the new name:

Unnamed Package

Figure 4

Component definition creation:

Within the definition package (Component Diagram) you will create the various components that make up the system. This is done by selecting the Component tool in the palette (see Figure 5). Once the tool is selected create the Component symbol in the same way that the Package above was created. Configure the created component by first naming it. Select the component and right click. In the menu choose the "Rename" menu item (see Figure 3). In the dialog that appears enter a name that is descriptive of the role within the system for the component (see Figure 4). It is good practice to give every element in the model a good description. The description shall detail the role within the system of the Component. To add a description select the component and right click. Choose the Open With > Description Editor menu item (see Figure 6). In the editor that is opened enter a good description (see Figure 7).

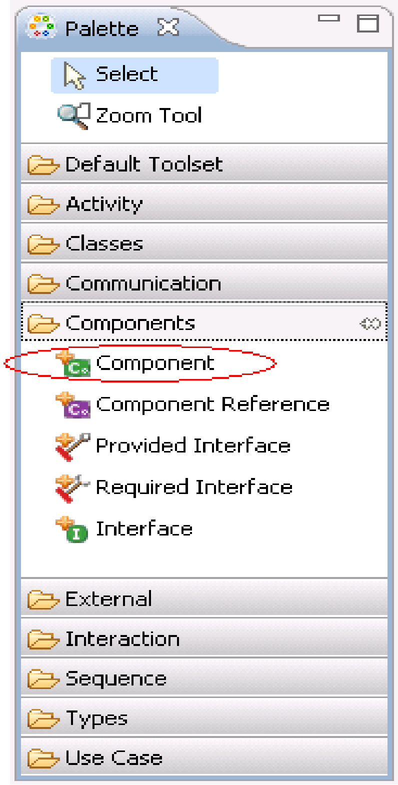


Figure 1.5

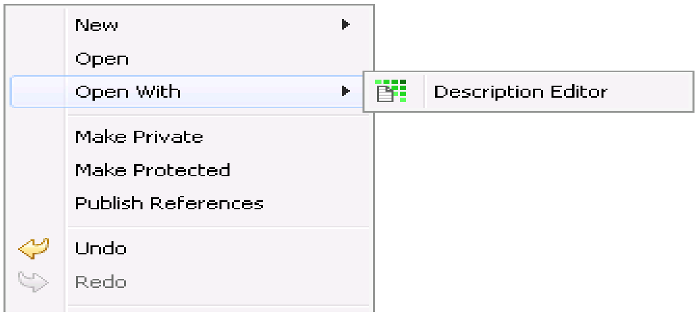
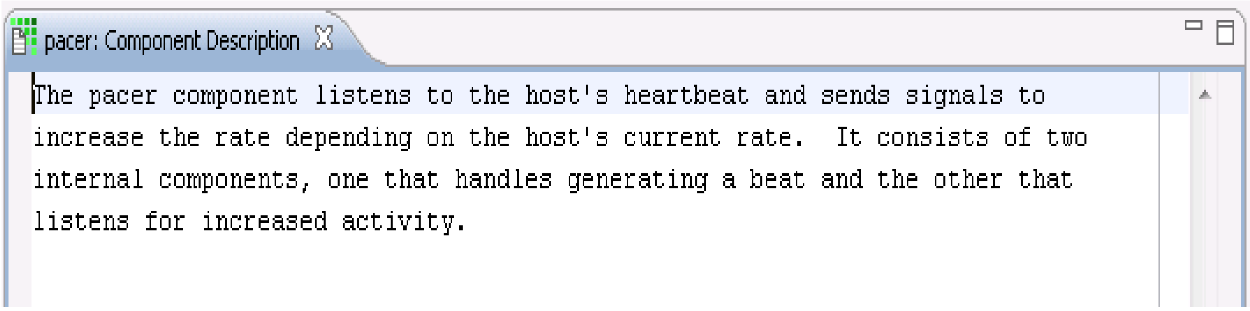


Figure 1.6



The pacer component listens to the host's heartbeat and sends signals to increase the rate depending on the host's current rate. It consists of two internal components, one that handles generating a beat and the other that listens for increased activity.

Figure 7

Interface creation:

Once the component definitions have been created it is time to consider the communication between them. As stated earlier, communications are configured by interfaces that define the necessary contracts. Interfaces will be created in their own Package. Create a new package according to the steps above relating to the creation of the Component Diagram. Name the package according to the role within the system of the components that will use the Interfaces. The Interfaces are created under this package. To create an Interface select the Interface tool in the palette (see Figure 1.7). Create the Interfaces according to the steps above relating to the creation of the Component Diagram. Give the Interfaces good names according to the communication role they play in the overall system. Give each of them a good description, which describes the same role (see Figures 1.6 and 1.7).

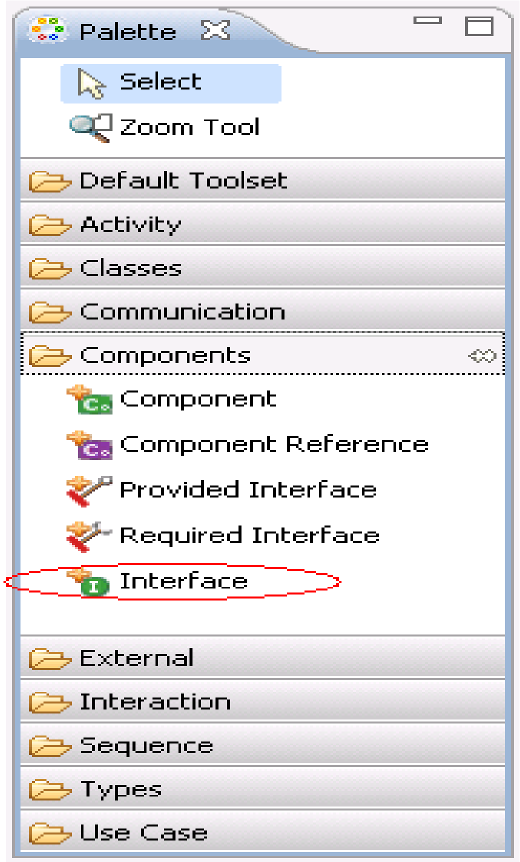


Figure 8

Interface Operation and Signal creation:

Once the Interfaces have been created it is time to create the operations and signals that define the available communication routes for the contract. Interface Operations are synchronous and return a value while Interface Signals are asynchronous and have no return. The steps for creating operations and signals are identical. To create them select the Interface and right click, then select the New > Operation or New > Signal menu item (see Figure 1.9).

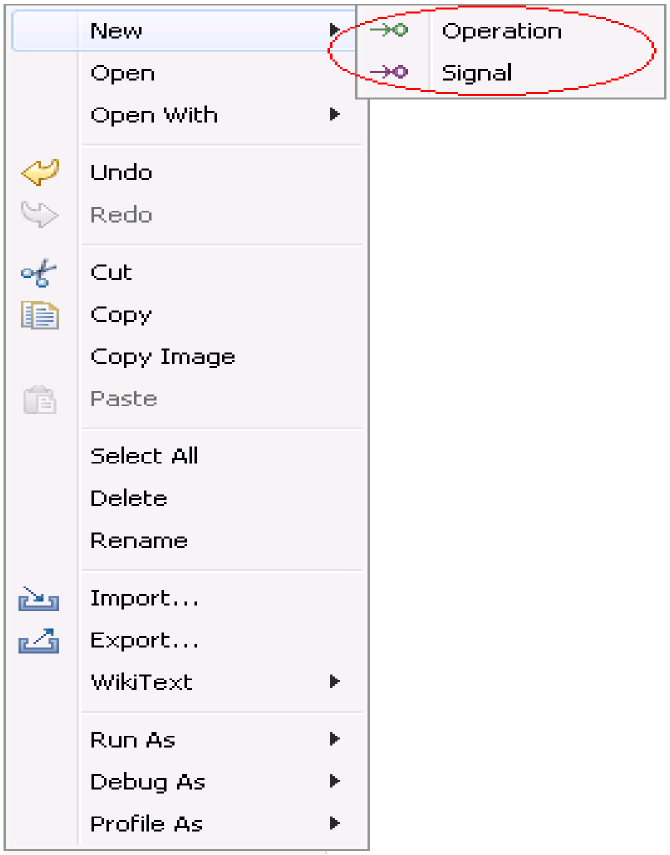
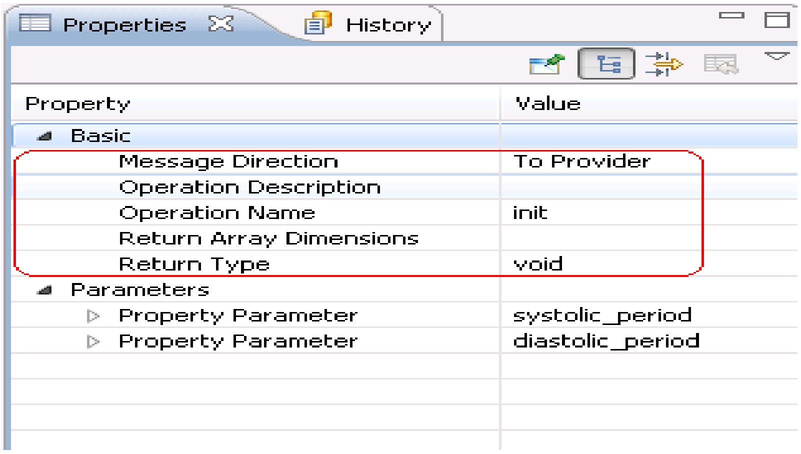


Figure 9

Now that the operations and signals are created it is time to customize them. Rename each by selecting them in Model Explorer and right clicking, then select the “Rename” menu item (see Figures 1.3 and 1.4). Give the interfaces and operations a good description as described for the Interface creation. Interface Operations can be further configured by setting the following attributes:

- Message Direction
- Return Array Dimensions
- Return Type

The above are set in the properties view (See Figure 1.10). The details of what each of these attributes mean are not within the scope of this document.



Property	Value
Basic	
Message Direction	To Provider
Operation Description	
Operation Name	init
Return Array Dimensions	
Return Type	void
Parameters	
Property Parameter	systolic_period
Property Parameter	diastolic_period

Figure 10

Interface Signals do not have any attributes relating to the return value, as they do not return anything. Therefore only the Message Direction needs to be configured.

Provided and Required Interface creation:

Each Component created must provide or require a certain interface to allow a contract. If a Component provides an Interface within a contract, it will be called upon to deliver or respond to such communication. The opposite is true when a Component requires a service in a contract. Once the provided and required services are determined for each Component it is time to create the provided and required interfaces. These are created in the same way. Select the proper tool, either the Provided Interface or Required Interface tool (see Figure 1.11).   Once the proper tool is selected left click on the desired Component that will use the Interface, then drag the mouse to the desired location and release the mouse button.

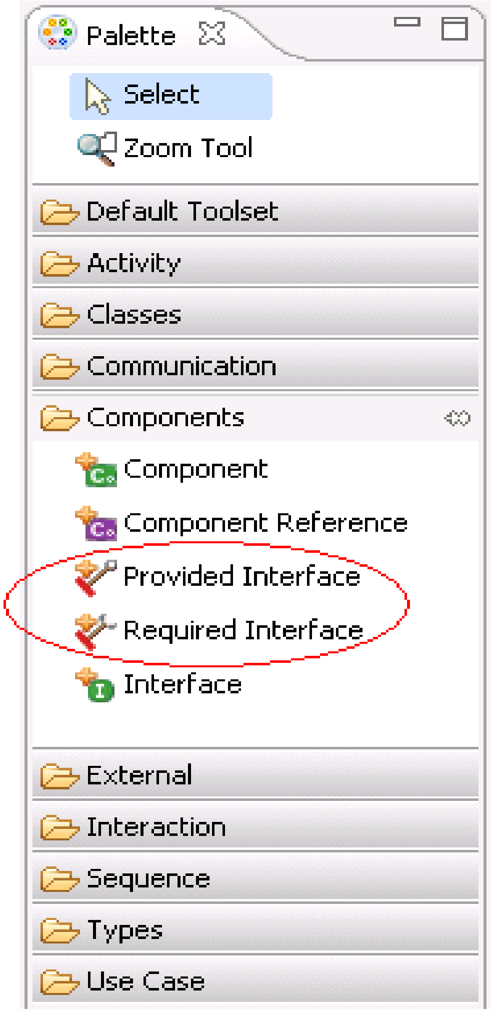


Figure 11

Formalization of provided and required interfaces:

Once the communication requirements are determined it is time to formalize the provided and required interfaces. This is accomplished by selecting the provided interface or required interface and right clicking. Next select the Formalize… menu item and choose the Interface that defines the required contract in the dialog that appears (See Figures 1.12 and 1.13).

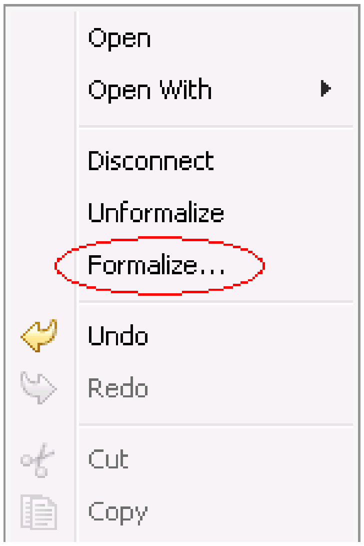
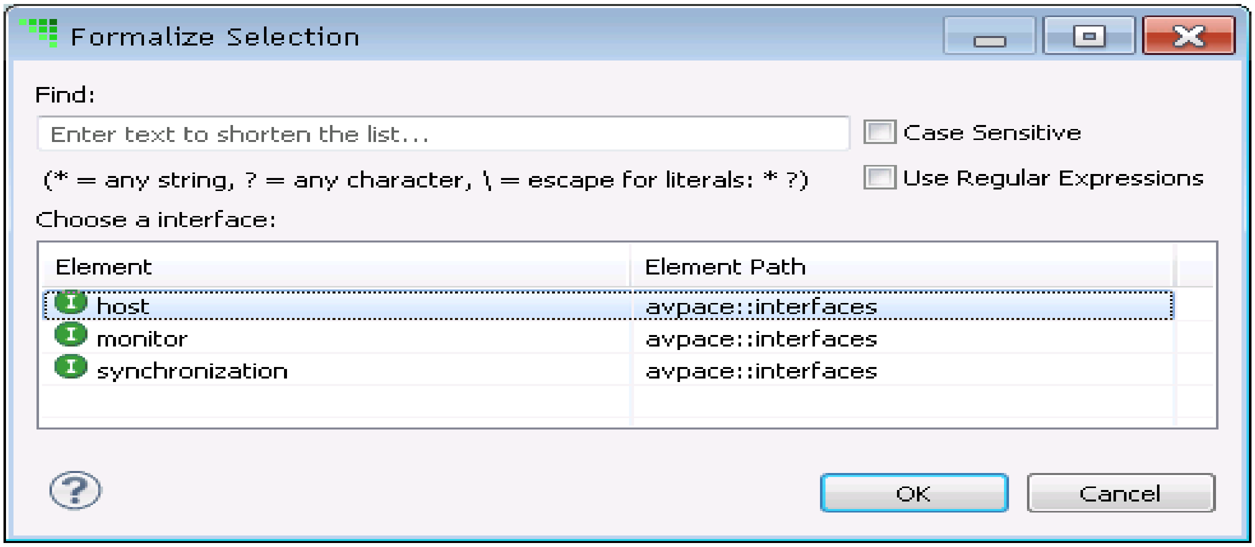


Figure 12


Formalize Selection

Find:

Enter text to shorten the list...

(* = any string, ? = any character, \ = escape for literals: * ?)

□ Case Sensitive

□ Use Regular Expressions

Choose a interface:

Element	Element Path
host	avpace::interfaces
monitor	avpace::interfaces
synchronization	avpace::interfaces

Figure 13

Component Reference creation:

Once the definitions are created it is time to fulfill the contracts among the various Components. Create another Package at the same level as the definition package. Give the package a name based on the particular usage that the contracted Components will fulfill. In the Package create a Component Reference for each Component that will communicate with each other. This is done by first selecting the Component Reference tool, the creating the symbol in the same way that the Package was created. Now select the Component References and right click. Select the Assign Component… menu item (see Figure 13). In the dialog that appears choose the Component that will be referenced (see Figure 14). The Component References will now show the path to the referred to Component and will have the pre-defined provided and required interfaces. To complete a contract with two Component References select one of the provided or required interfaces, and drag its end to the other provided or required interface (see Figure 15).

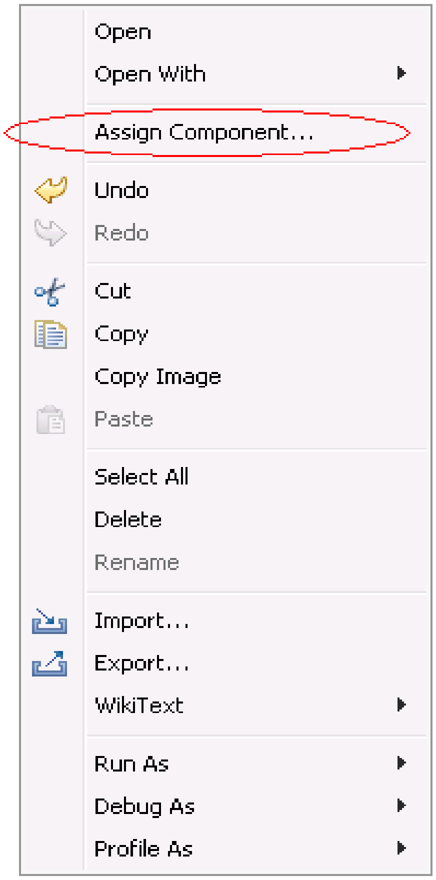
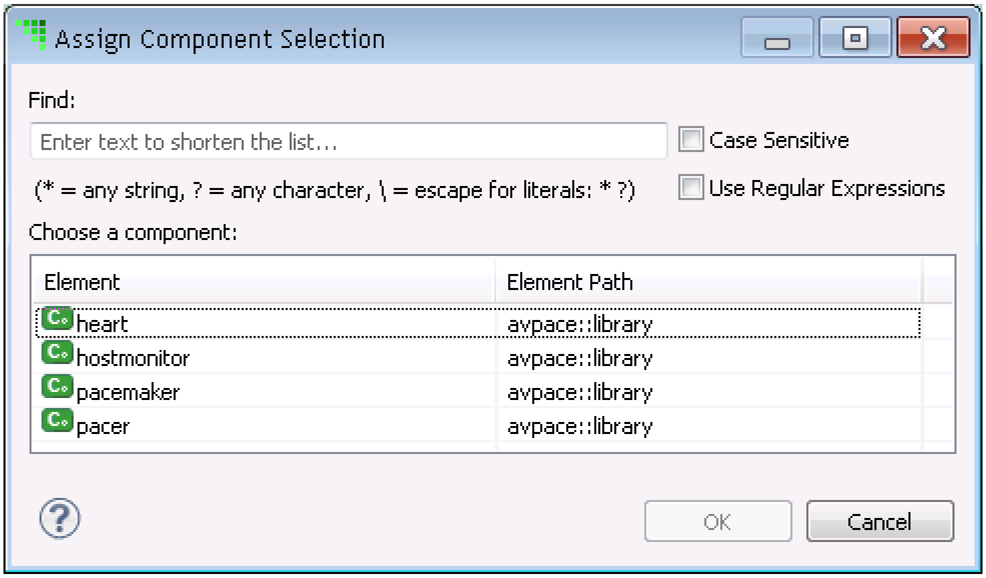


Figure 13



Assign Component Selection

Find:

Enter text to shorten the list...

□ Case Sensitive

(* = any string, ? = any character, \ = escape for literals: * ?)

□ Use Regular Expressions

Choose a component:

Element	Element Path
heart	avpace::library
hostmonitor	avpace::library
pacemaker	avpace::library
pacer	avpace::library

Figure 14

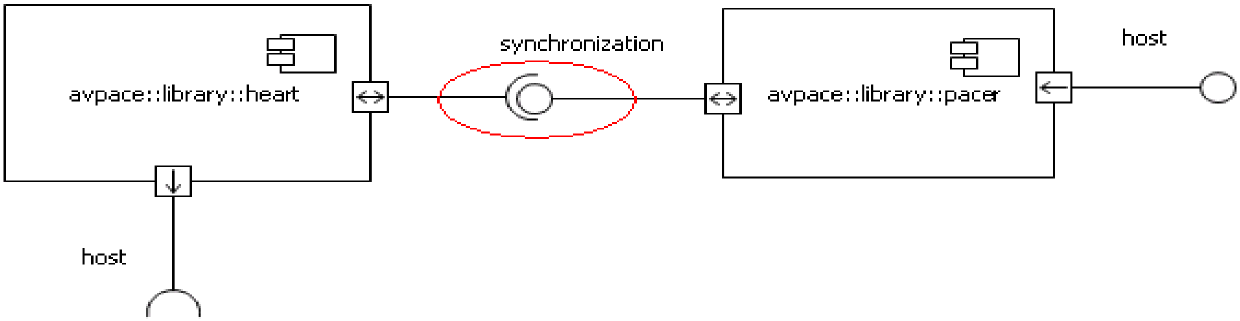


Figure 15

Component References that have been assigned can be unassigned by right clicking and choosing the Unassign menu item. Note that when unassign all interface references will be removed and any contract made will be lost. Component References that are assigned may also be re-assigned. This is accomplished by right clicking on a Component Reference and choosing the Assign Component… menu item. The rule for re-assignment is that the new Component must at least fulfill the same Interface set. The new Component may support additional Interfaces.

Interface references that have been connected can be disconnected by right clicking on either end and selecting the Disconnect menu item. Once done each interface reference end can be freely moved around and reconnected with other interface references.

Internal structure creation:

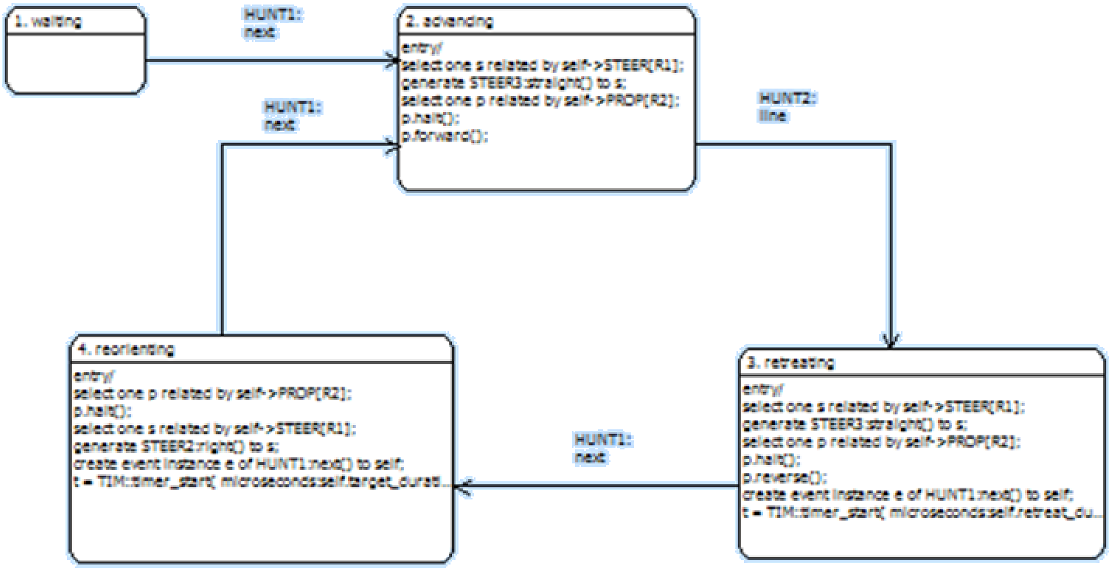
The internal structure of a Component is created by first opening the Component definition. To open the Component's internal structures double click on the Component definition in either Model Explorer or the Component definition diagram. The elements that define the internal structure are created within the Component symbol that pre-exists. The same elements that are created on the Component definition or Component reference diagram can be created on the internal structure diagram. See the steps above for creation of those elements. In addition to those elements there are two others that can be created on the internal structure diagram. These are the Package element and the Delegation element. The Package element allows for creation of all BridgePoint element types to be part of the internal structure. Package details are not contained in this document. The Delegation element is a connection between outside communication and inside elements. The pre-existing Component symbol will have all of the provided and required interfaces pre-existing as well. To create a Delegation the internal structure must contain a Component or Component Reference that uses the same formal Interface type. Select the Delegation tool and left click on either the outer interface reference or the internal one, then drag the mouse and release the mouse button at the other interface reference. Another way that communication can be delegated is through transition assignment. This is not covered in this document as the delegation is not visible on the Component diagram.

Recommendations

When creating Component definitions consider the subject matter that can be separated within the system. That is the subject matter that can stand alone. When the system is partitioned into Components it allows for component replacement. This provides a good way for testing as well as multiple system deployments.

Always use the Component reference diagram for making the contracts, otherwise component replacement will not be supported easily.

State Diagram



```
entry/
select one s related by self->STEER[R1];
generate STEER3:straight() to s;
select one p related by self->PROP[R2];
p.halt();
p.forward();
```

```
entry/
select one s related by self->STEER[R1];
generate STEER3:straight() to s;
select one p related by self->PROP[R2];
p.halt();
p.reverse();
create event instance e of HUNT1:next() to self;
t = TIM::timer_start( microseconds:self.retreat_du...
```

```
entry/
select one p related by self->PROP[R2];
p.halt();
select one s related by self->STEER[R1];
generate STEER2:right() to s;
create event instance e of HUNT1:next() to self;
t = TIM::timer_start( microseconds:self.target_durati...
```

A State Machine diagram documents the lifecycle of a class using a combination of State boxes joined by Transitions between them. An instance of a class is said to be driven between the States of its lifecycle based on the sequential arrival of Events. Events may carry Parameters which provide data to the behavior specified in the Actions associated with Transitions and States. Actions are considered to be carried out while transitioning from one State to the next and on entry to a new State.

Things to Know

Prior to beginning a State Machine Diagram it is important to have all information related to

Individual State

Events

Signal

Transitions

Parameter

Action

When to Use

State Machine diagrams are constructed for many, but not necessarily all Classes in a design. Use a State Machine diagram if instances of a given Class exhibit some sense of history. That is, it responds differently to the same Event depending on what has gone before. Less commonly, State Machines are constructed for the Class itself as opposed to instances of the class. One use for Class based State Machines is to manage contention for a resource, see [xtUML, section 13.2]. Another useful feature of Class State Machines is that Signals may be assigned to Transitions in these.

Creation

Create a new State Machine diagram for the Class by right clicking on it and selecting New > Instance State Machine. A small symbol will appear on the Class signifying that an Instance State Machine exists for it. Double click the Class to open the diagram. To create a class based State Machine, use New > Class State Machine.

State Symbol

Create a State symbol by locating the State tool in the palette and select it. Now move to the diagram and choose a suitable space on the sheet. Click on the left mouse button and drag the mouse until the outline is the size you want and release. The State symbol will be drawn. Click on the right mouse button and rename the State, giving it the name identified during the State Machine development procedure. Right click on the State and choose Open > Description. Enter the description of the State into the editor and save it using File > Save, clicking the disk icon on the top tool bar or pressing <Ctrl+S>.

Event

Create a new Event by clicking the right mouse button and choosing New > Event. No symbol is shown for an Event until it is assigned to a Transition. Use the Model Explorer view to locate the new Event, right click on it and rename the Event using the name identified for it during the State Machine development procedure.

Add Event Parameter

Locate the Event in the Model Explorer. Click on the right mouse button over it and select New > Parameter. Select the new Parameter and choose Rename..., to give it a name and Set Type..., to change its type from the default integer setting.

Transition

Select the Transition tool in the palette. Click the left mouse button over the starting State and drag across to the ending State. You do not need to click accurately on the edge of the State symbols, draw from center to center. The tool will clean up the drawing automatically. Sometimes, you will want to show a transition returning to the same State as it left, to draw this make sure you begin the drawing action near the center of the State. Now drag the mouse a little without leaving the boundary of the State. The tool will detect what you want to do and draw the looped back Transition symbol for you.

Creation Transition

A Creation Transition is one which is sent to the Class, in response to which an instance is created. To draw one, choose the Creation Transition tool from the palette. You can either start on white space and drag the transition into the initial State or start in the State and stop dragging over the canvas background. Either way the tool will draw the Transition going into the target State.

Assign an Event to a Transition

Click the right mouse button over the Transition you wish to assign to. Choose Assign Event... A dialog appears allowing selection of the Event. Sometimes, there may be no Events to choose from. This is for one of the following reasons:

- No Events have been created yet for the diagram.
- All Events created on the diagram are already assigned to outgoing Transitions from the starting State.
- The Events already assigned on Transitions to the ending State carry a data set for which no remaining available Event matches.

In the case of Class based State Machines, another menu entry is available on a right mouse click; Assign Signal. Using this is identical to assigning an Event.

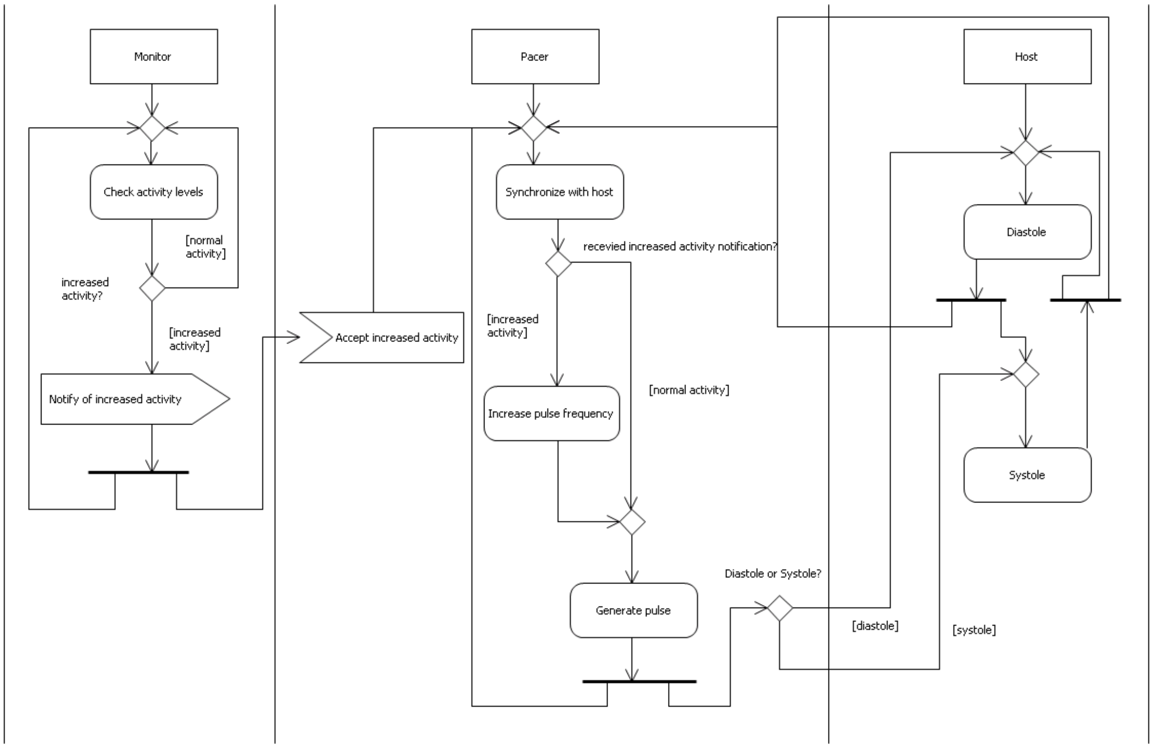
When all States have been visited, switch to the State Event Matrix view by clicking on the tab below the State Machine diagram. The tool switches to a new editor that shows the State/Event matrix. For each cell in the matrix that is marked Can't Happen, specify one of the three possibilities:

- The Event really Can't Happen in the corresponding State
- The Event _can_ happen but does not cause any change of State
- A Transition has been missed

No action is required in the first case. In the second case, click on the cell in the matrix and select Event Ignored. The third case should not occur if the procedure for creating the State Machine diagram has been followed correctly.

See the procedure documentation for the action required.

Activity Diagram



The Activity Diagram depicts the flow of control for an activity. The documented activity consists of several sub-activities that are involved in the overall flow.

Things to Know

Prior to beginning an Activity Diagram it is important to locate all information related to

Accept Event Actions

Actions

Object Nodes

Initial Nodes

Flow Final Nodes

Activity Edges

Accept Time Event Actions

Send Signal Actions

Decision/Merge Nodes

Activity Final Nodes

Activity Partitions

Fork/Join Nodes

When to use

An Activity Diagram can be used to document a business workflow or a software workflow for a given activity.

When documenting a business workflow the activities and flows are real world elements that are carried out by parts of the organization. Use an Activity Diagram to produce a clear picture of the overall flow for situations that must occur within the documented workflow.

When documenting a software workflow the objects, actions and control flow can be mapped to parts of the system. Use an Activity Diagram to produce an overview of the control flow that will aid in the creation of the real software elements in the system. The Activity Diagram can additionally be used after the fact (after the real software elements have been created) to provide a clear picture of the workflow for a given situation.

Prior to beginning an activity diagram, it is important to know

1. The involved participants.
2. The control flow among the participants.
3. Various paths involved within the control flow.

The resulting document is living and may change over time. In both the business model and software case the document can change as the project progresses and further detail is learned.

A well written Activity Diagram will clearly depict the flow of control among the set of participants. It will be easy to follow the flow. If the documented flow becomes large the flow will grow harder to read. If this occurs consider stepping back and refactoring the document into smaller flows that can be referenced. When breaking up a large activity use a package hierarchy and naming scheme to allow easy navigation for the reader. An example follows:

Descriptive Activity Name (Package)

|_ Descriptive Sub Activity Name Step 1 (Package)

|_ Descriptive Sub Activity Name Step 2 (Package)

Usage

Generic:

In the generic case the goal of the document is to better understand the general workflow. This is true for both the business model and software situations. This document may map to real participants, but may just depict ideas. There is no formal way to map objects, activities and flow in the BridgePoint tool. To achieve mapping take consideration when naming the various pieces of the document.

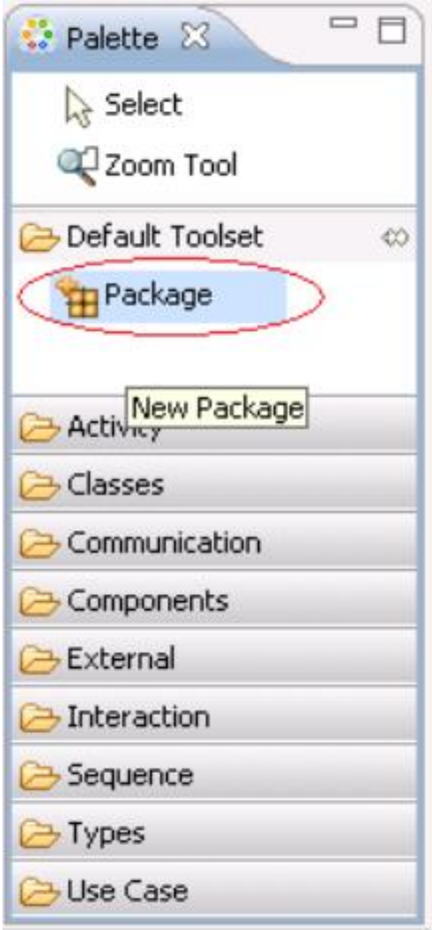
Precise:

In the precise case the workflow participants and control flow is fully understood. As stated above there is no formal way to achieve mapping. As is true with the generic case use good naming procedures to produce a mapping.

Steps to creating an effective Activity Diagram

Diagram creation:

Once the location has been determined, locate the palette view (see Figure 1.2 below). In the palette view select the Package tool (as shown in Figure 1.2). On the editor page click the left mouse button and drag a distance. A marquee will be drawn that indicates the location and size of the graphical symbol to be created. Once the symbol is at a desired location and size let go of the left mouse button. At this point a new Package has been created. Right click on the Package graphical symbol and select “Rename” (See Figure 1.3 and 1.4 below). In the window that is opened enter a good name for the package. This name should reflect the situation that control flow is being captured for.



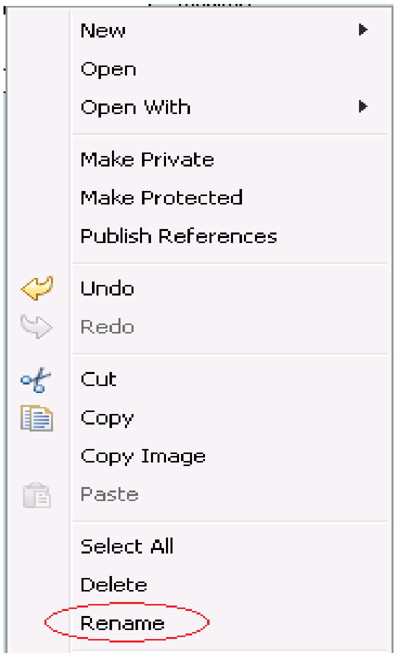
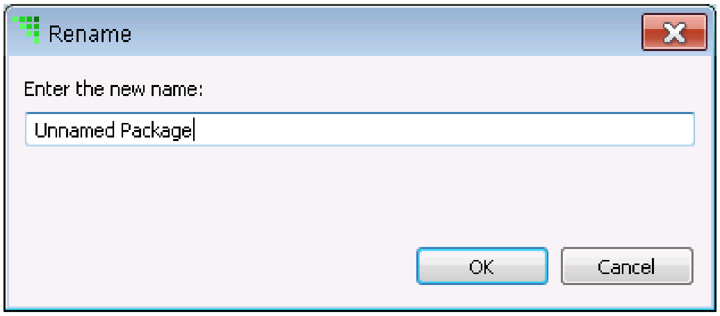


Figure 1.2

Figure 1.3



Enter the new name:

Unnamed Package

Figure 1.4

Action creation:

As with diagram creation, the proper tool must be selected. As stated in the Elements section there are many Activity elements. They are broken up into actions, nodes, and flow (Activity Edge in BridgePoint). Select the action tool that is of interest (see Figure 1.5 below). Once selected proceed to draw the symbol for the action graphical element (see steps in package creation for details). At this point the newly created action can be configured. Start by giving the action a name, remembering to take consideration of the name used. Select the action and right click, then select the "Rename" menu item (See Figure 1.3). In the dialog that appears enter the desired name (See Figure 1.4). It is a good idea to enter a description for the element created. The description shall describe the role of the element within the documented workflow. To set the description right click on the element and choose the Open With > Description Editor menu item (See Figure 1.6 below). Enter the desired description text and save the changes (See Figure 1.7 below).

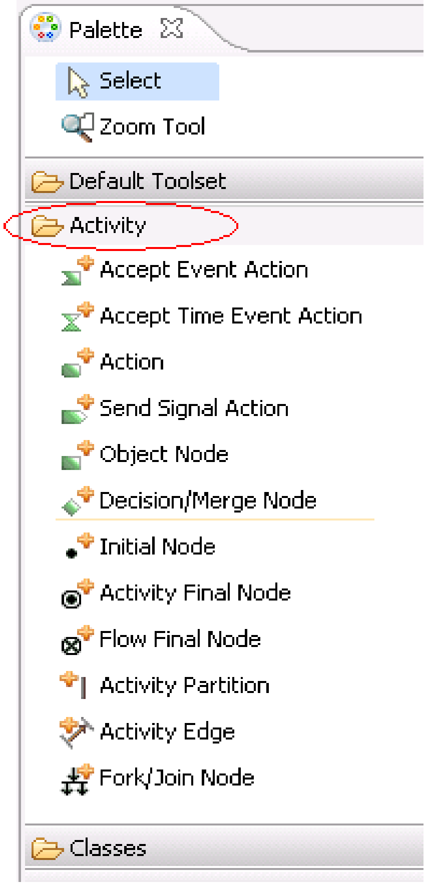


Figure 1.5

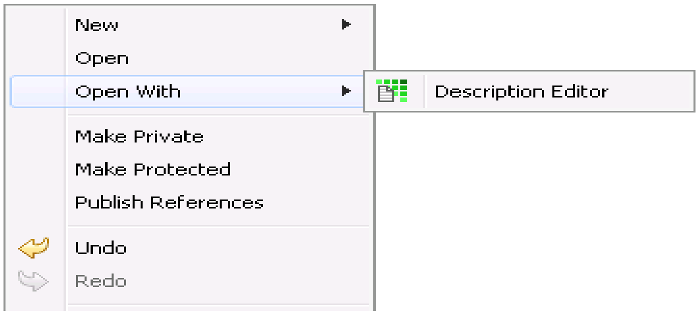
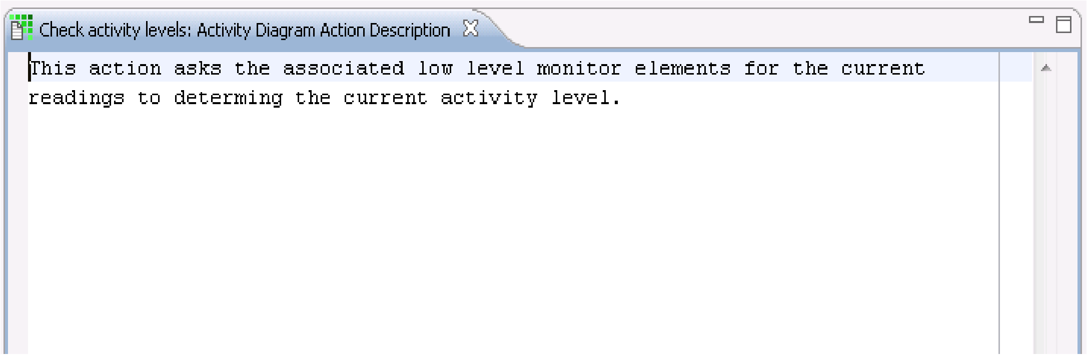


Figure 1.6



```
This action asks the associated low level monitor elements for the current
readings to determing the current activity level.
```

Figure 1.7

There are four types of action that can be created in the Activity Diagram. These are:

1. Action
2. Accept Event Action
3. Accept Time Event Action
4. Send Signal Action

It is not within the scope of this document to describe the meaning of each of these. Creation of each action type is identical to the prescribed procedures above.

Node creation:

Select the node tool that is of interest (see Figure 1.5 above). Once selected proceed to draw the symbol for the mode graphical element (see steps in package creation for details). At this point the newly created node can be configured. Start by giving the node a name, again remembering to take consideration of the name used. Select the node and right click, then select the “Rename” menu item (see Figure 1.3). In the dialog that appears enter the desired name (see Figure 1.4). It is a good idea to enter a description for the element created. The description shall describe the role of the element within the documented workflow. To set the description right click on the element and choose the Open With > Description Editor menu item (see Figure 1.6 above). Enter the desired description text and save the changes (see Figure 1.7 above).

There are six types of node that can be created in the Activity Diagram. These are:

1. Object Node
2. Decision/Merge Node
3. Initial Node
4. Activity Final Node
5. Flow Final Node
6. Fork/Join Node

As mentioned above the meaning of each of the above elements is not within the scope of this document. The creation of each node is nearly identical to the steps above. The nodes that behave differently are described below.

The Decision/Merge Node may not always have a name set. In most cases when this element is used as a Merge the name is omitted. When the element is used as a Decision, the name generally captures the question that the decision is made against.

The Initial Node, Activity Final Node, and Flow Final Node are never named. In addition the Initial Node should never have a flow drawn to it, only from it. The opposite is true of the Activity Final Node and the Flow Final Node.

The Fork/Join Node is not a shape symbol like the other nodes; it is treated as a connector in BridgePoint. This allows for easier orientation configuration, where the node can be drawn horizontally or vertically. This means that the movement and resizing behavior is rather different than the other nodes. Similarly the creation is much different than the other nodes. To create a Fork/Join Node select the tool in the palette view. Once selected left click the mouse at the desired start location, then drag and release the left mouse button at the desired end location. This element can be named to describe the fork or join role within the workflow.

Flow creation:

As with the other elements the proper tool must be selected. In BridgePoint this is the Activity Edge tool. Once selected left click the mouse at the desired start element, then drag and release the left mouse button at the desired end element. The Activity Edge can be named, but is commonly omitted. The naming is generally used to indicate a guard condition where the flow has left a Decision/Merge Node, which is named after the decision made. All other elements, with the exception of the Activity Partition, are valid start and end elements for the Activity Edge. The tool will not allow multiple incoming or outgoing Activity Edges to the same action or node with the following exception. Only the Decision/Merge Node and the Fork/Join Node may have multiple entries and exits. These two nodes are present for this exact case.

Partition creation:

As with the other elements creation begins with selecting the proper tool. In this case the Activity Partition tool. Once selected the Partition is created in the same way as the Fork/Join Node. Left click at the desired start position, then drag and release the left mouse button at the desired end location. Partitions can be named if desired.

Recommendations

As with any programming language, UML models should be factored into the smallest amount of data that makes sense. It is easier to read and understand a diagram (or program code) when the subjects within the system are factored into many small functional parts. This prevents the diagram from growing large to a point where readability is hindered.

Always include descriptions, for the elements, that describe their role in the documented workflow.

If a diagram is growing to the point where the flow is hard to follow, take a step back and consider the possibilities for refactoring. Readers can always be referred to a set of documents to help describe a larger procedure.

Implementation Targets

The Model Translation (also known as Code Generation) step is where a model compiler is used to perform translation of UML to target code (C, C++, Java, SystemC, Ada …). Mellor and Balcer provide an introduction to the model compilers, and their operation in *Executable UML* [1]. As they state:

> **[Y]ou must choose how to compile your Executable UML models based on the performance requirements and the environment of your application. From this information, you can select a model compiler that meets your needs, compile the models, and deliver the running system.**
>
> **You can buy an existing model compiler as a "design-in-a-box." Alternatively, you can modify an existing model compiler or even build your own from scratch**

This chapter discusses the capabilities of several off-the-shelf model compilers available from Mentor Graphics for C, C++, and SystemC. Also included are instructions on how to install these commercial model compilers or one that you have created.

Embedded C / C++

The BridgePoint embedded C model compiler, known as MC-3020, is available in two forms. The binary (non-modifiable) version of MC-3020 is included in every BridgePoint installation. An equivalent source (user-modifiable) version is available as a separate product from Mentor Graphics, and provides a starting point for the development of a corporate of project-specific model compiler. The embedded C++ model compiler is available in source form from Mentor Graphics.

These model compilers generate optimized C or C++ source code that is suitable for use in all manner of embedded environments. The generated application contains a small, concurrent, and highly-efficient event-driven architecture that can run directly "on the iron" or on top of a real-time operating system (RTOS). The C and C++ model compilers are reliable and proven tools to generate the application implementation in C or C++.

SystemC

The BridgePoint SystemC model compiler is used to translate the xtUML model into an implementation suitable for execution in a SystemC simulator. The SystemC model compiler can generate code that is compatible with standard SystemC ports as well as Transaction-level Modeling extensions.

SystemC is useful to build application models that express both hardware and software blocks, and the interfaces between them. BridgePoint SystemC is used for:

- **Architectural exploration.** At the beginning of the development process, you must answer questions such as: How much processing capability do I need? How much RAM is required? What are the power requirements? What blocks must be implemented in hardware? What blocks can be implemented in software? System-level models created using BridgePoint and simulated in SystemC help answer these questions.
- **Defining a firmware functional verification platform.** With the ability to model and execute the functionality of hardware and software blocks, the SystemC simulation can be used to create a test infrastructure that verifies proper operation of the software in the context of the hardware.
- **Creating a Hardware Abstraction Layer.** BridgePoint's ability to model and simulate both hardware and software makes xtUML an ideal environment to model software that abstracts the functionality of the hardware and defines the interface points to the hardware blocks.
- **Hardware/software co-design.** Multiple BridgePoint model compilers can be used against a single project. This capability allows you to translate SystemC blocks in one pass, and embedded C target code in another pass. Because the xtUML model is platform independent, BridgePoint enables you to retarget your application from simulation to implementation code according to your needs. At the same time the software is designed and refined, the hardware undergoes the same design and refinement until it, too, is ready for implementation. In this way, the tool helps drive the implementations of the blocks.

Preparing to Build a Project

To translate a project into useful implementation code, there are several prerequisites to attend to. These include:

- A model compiler must be installed and ready to use. Specific instructions how to do this are found in the following section, Installing BridgePoint Model Compilers.
- The model itself needs to have some specific pieces in place. The model must have a package containing the system to be translated. Typically, this is a package that contains a number of component references whose interfaces are connected to each other.
- The model must contain a domain function that serves to bootstrap the application execution. This initialization function may create key classes and active state machines or send interface messages to other components that jumpstart their initialization and execution.
- The project must have marking information specified that is used during the model translation. Only a few marks are essential to create a running application, but many are available to fine-tune the code generation process. Marking is discussed in more detail in the following sections about specific implementation targets.
- The correct eclipse project builders must be enabled. Builder configuration is also discussed in more detail in the documentation about specific implementation targets.

Installing BridgePoint Model Compilers

BridgePoint contains a built-in model compiler that generates fast and small C code for embedded systems. No additional licensing or configuration is necessary to get started right away building models with this model compiler.

However, you may wish to use a different commercially available model compiler, or even write your own. To use one of these non-built-in model compilers, BridgePoint must be properly licensed and configured. A floating or node-locked license for the appropriate model compiler is required. Contact local IT or Mentor Graphics Customer Support for assistance installing and configuring licenses.

A Note about Environments

The BridgePoint model compiler uses a Windows-based tool during the translation process. Linux users must install Wine [2] in order to run the model compiler. See Linux guide in BridgePoint Help for more information.

The modifiable C model compiler and the SystemC model compiler are distributed separately from the BridgePoint application installer, typically as an archive file. Decompress the archive into:

- `<BridgePoint_install_directory>/eclipse_extensions/BridgePoint/eclipse/plugins/com.mentor.nucleus.bp.mc.mc3020_<version>/mc3020`
-

This action creates an `arc/` subdirectory.

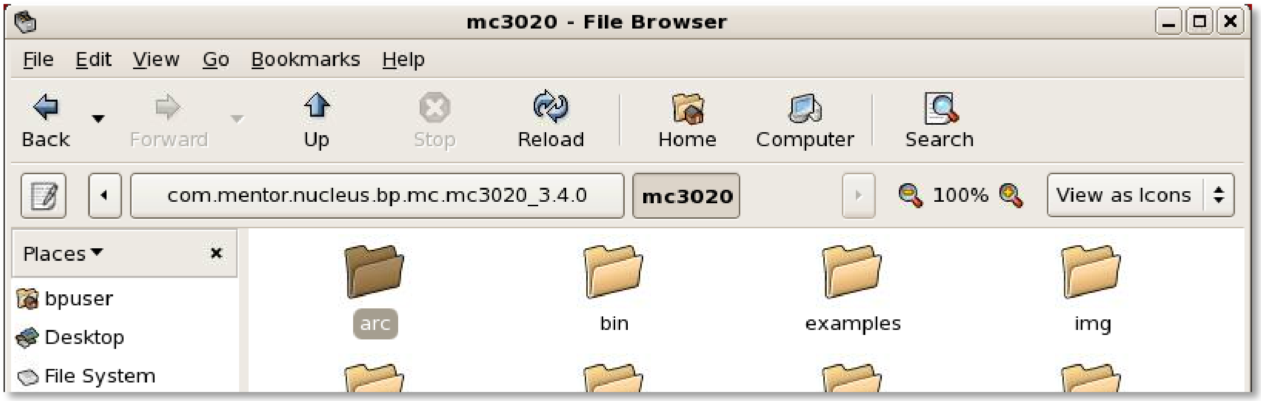


Figure 10

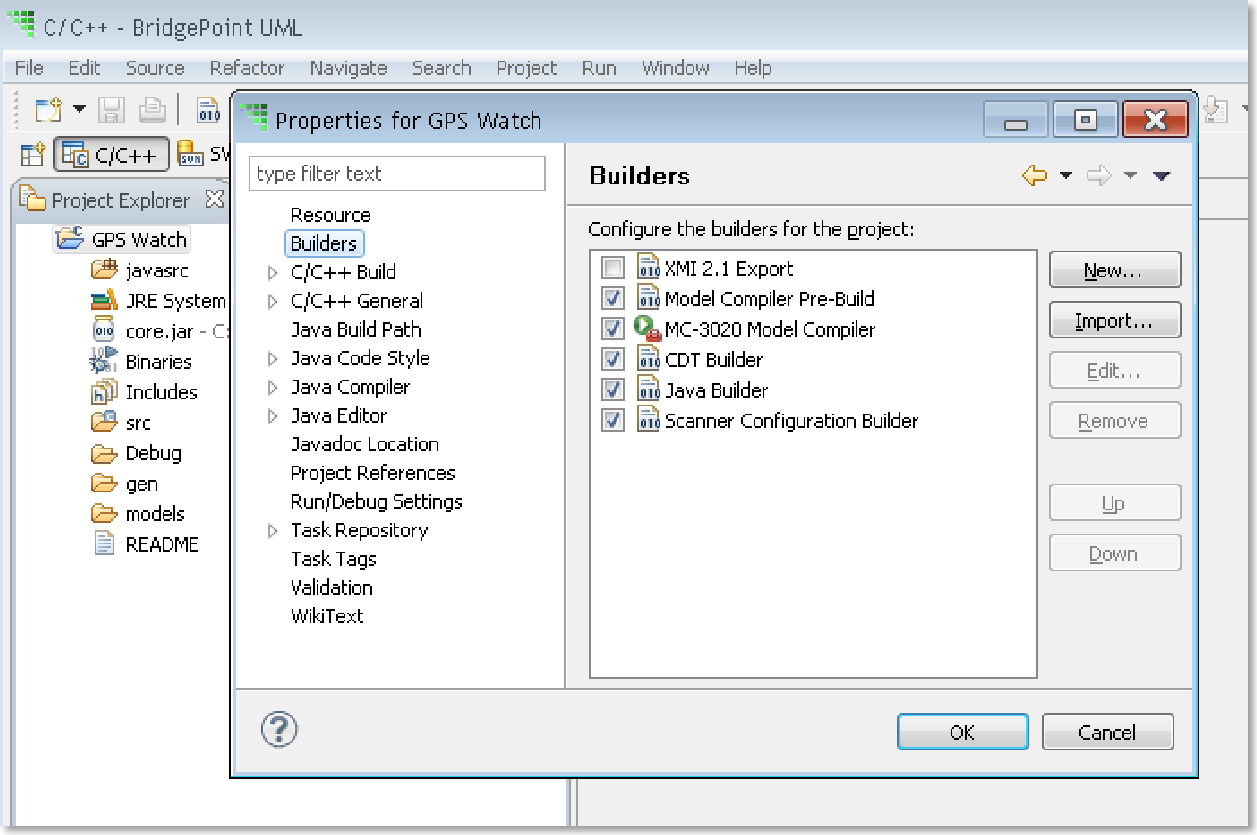
If you wish to build your own model compiler, you must acquire the proper license from Mentor Graphics and configure the license in your BridgePoint installation. With the license in place, it is then simply a matter of creating and `arc/` folder as above and populating the folder with your model compiler archetypes.

Implementing a BridgePoint Model in Embedded C

This section describes how to translate the model into a C implementation for embedded targets and execute the application in the host environment.

Generating C Source Code

BridgePoint performs translation in a sequence of steps using eclipse builders:



Builders

Configure the builders for the project:

- ☐ XMI 2.1 Export
- ☑ Model Compiler Pre-Build
- ☑ MC-3020 Model Compiler
- ☑ CDT Builder
- ☑ Java Builder
- ☑ Scanner Configuration Builder

Figure 11

The first step is known as "Model Compiler Pre-Build." In this step, the model data is assembled together in a form that is usable by the model compiler itself. Once the data is prepared, the model compiler /code generator processes the data using its mechanisms and archetypes. The output of this step is the implementation code. Lastly, the implementation code is compiled by the eclipse CDT, which is built into BridgePoint. The built-in code compilation step can be turned off in order to use a target compiler of your own choosing at a later stage in your development process. As shown in Figure 1, the GPS Watch sample model includes a Java Builder as well to build the Java-based UI included in this example.

Markings

BridgePoint uses marking information to provide “knobs and dials” for fine tuning the code generation process. This information is contained in `.mark` files located in the `gen/` folder. See the Marking section of the BridgePoint Model Compiler in the Help system for extensive discussion of this topic.

You can get started quickly by adding the following marks to `gen/system.mark` and `gen/domain.mark` in your project:

system.mark

```
.invoke MarkSystemConfigurationPackage( "<your system package>" )
```

domain.mark

```
.invoke MarkInitializationFunction( "*", "<your init function>" )
```

Building the Project

Once the markings are in place, build the project by selecting “Build Project” in the project’s context menu as shown in Figure 2. Informational output from the model translation process is placed in the Console view. The generated code is put into the project’s `src/` folder.

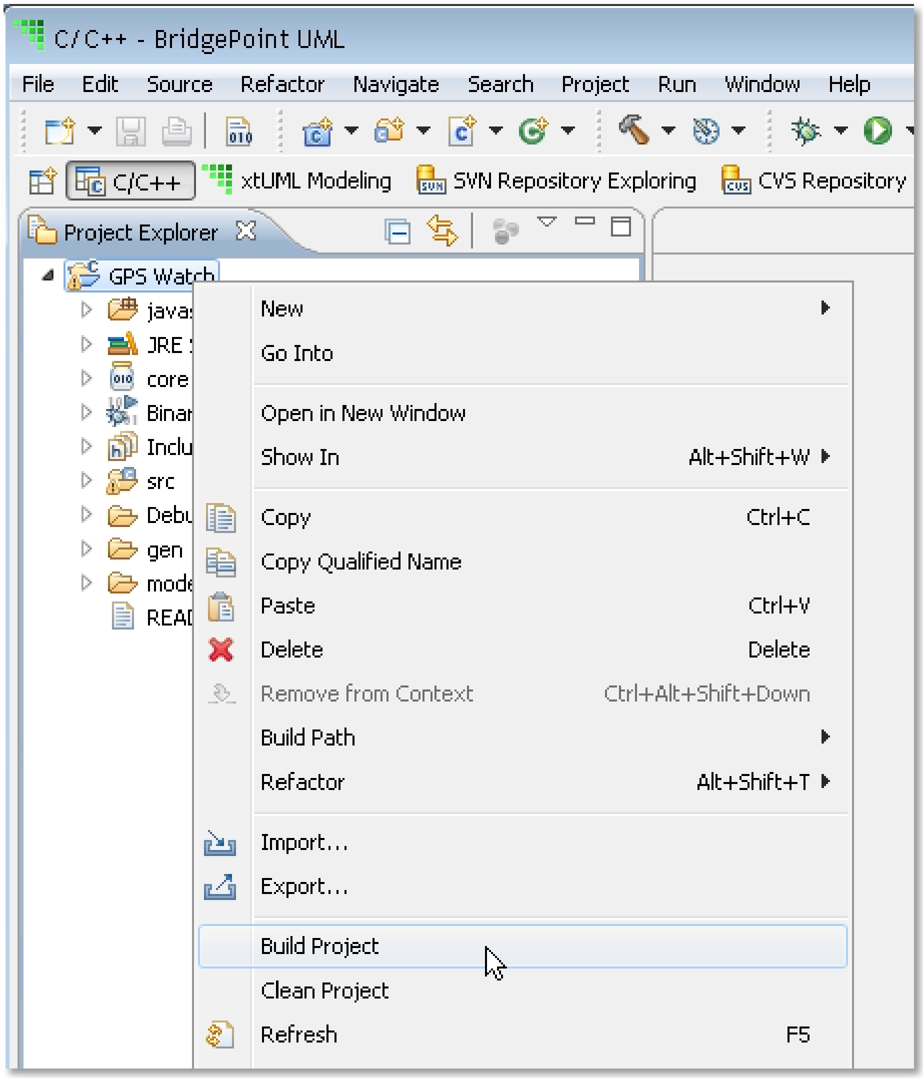
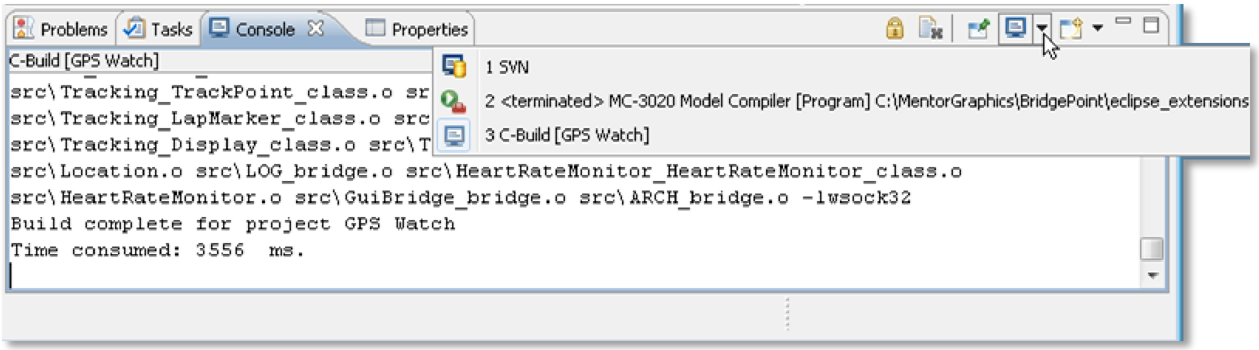


Figure 12

Both the BridgePoint model compiler and the eclipse CDT compiler output messages to the Console view as they process the model and implementation code. The console view treats the output separately. Use the Console view display switcher as shown in Figure 3 to choose the output you want to view.


Problems
Tasks
Console
Properties
1 SVN
2 <terminated> MC-3020 Model Compiler [Program] C:\MentorGraphics\BridgePoint\eclipse_extensions
3 C-Build [GPS Watch]

```
C-Build [GPS Watch]
src\Tracking_TrackPoint_class.o sr
src\Tracking_LapMarker_class.o src
src\Tracking_Display_class.o src\T
src\Location.o src\LOG_bridge.o src\HeartRateMonitor_HeartRateMonitor_class.o
src\HeartRateMonitor.o src\GuiBridge_bridge.o src\ARCH_bridge.o -lwsock32
Build complete for project GPS Watch
Time consumed: 3556  ms.
```

Figure 13

Overriding Generated Output and Implementing External Entities

BridgePoint application models are not restricted to live in a world isolated from other software. xtUML models can integrate with domains outside the model such as in-house legacy software and third-party tools. These outside domains are modeled as External Entities (EEs).

Executable UML[1] details how to properly create and communicate with External Entities in the xtUML model.

The BridgePoint model compiler generates code for all the external entities specified in the model. When it is time to translate the model to an implementation, you can override the generated code with your own implementation of the external entity.

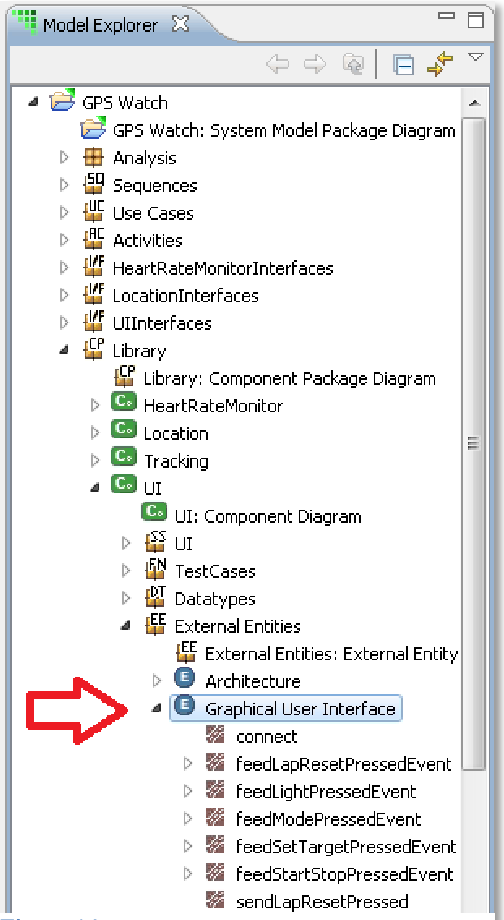
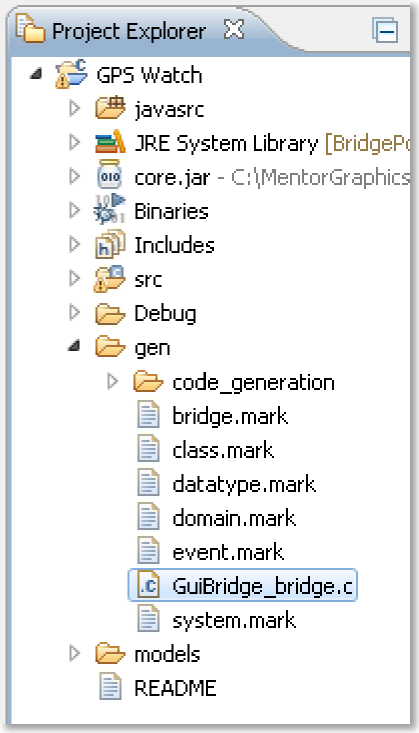


Figure 14

Figure 4 shows the Graphical User Interface (key letters GuiBridge) external entity of the GPS Watch project. This EE communicates with the GPS Watch GUI over a socket connection. The xtUML model does not contain native code to do this. The EE simply models the interfaces to this domain that the rest of the application needs to use.



```
Project Explorer
GPS Watch
  javasrc
  JRE System Library [BridgePo
  core.jar - C:\MentorGraphics
  Binaries
  Includes
  src
  Debug
  gen
    code_generation
    bridge.mark
    class.mark
    datatype.mark
    domain.mark
    event.mark
    GuiBridge_bridge.c
    system.mark
  models
  README
```

Figure 15

When the model compiler translates model elements, it always creates regular file names based on the element's key letters and the type of the element. For example, external entities are named `<key letters>_bridge.<extension>`.

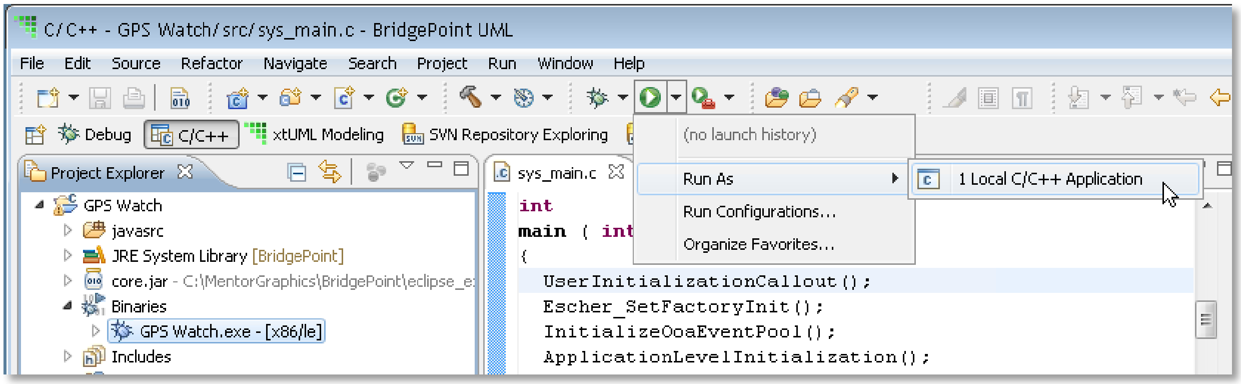
At project build time, the model compiler translates all the model elements, including the EEs. Then, at the end of the translation but before compilation, all source code files in the `gen/` folder are copied to the `src/` folder. Thus, the Graphical User Interface EE implementation-specific file shown in Figure 5 will overwrite the one generated by the model compiler in the `src/` folder.

It is good practice to let the model compiler provide the structure of the bridge implementation file(s). Meaning, don't try to create the bridge implementation in the `gen/` folder from scratch. Build the project and let the model compiler generate the skeleton data and in the bridge files. This will include the proper file layout and function prototypes with comments showing where to insert implementation-specific code. Copy the initial declaration and/or definition files from `src/` to `gen/`. From this point on, edit the details of the EE implementation in the files under gen/.

Running the Implementation Code

As discussed previously, the default configuration will compile the implementation code using CDT for the host platform. When the application is built successfully, it shows up under the project in the Binaries entry.

BridgePoint includes built-in facilities for running and debugging the application. Figure 6 shows how to run the application directly.



```
int
main ( int
{
  UserInitializationCallout();
  Escher_SetFactoryInit();
  InitializeOoaEventPool();
  ApplicationLevelInitialization();
```

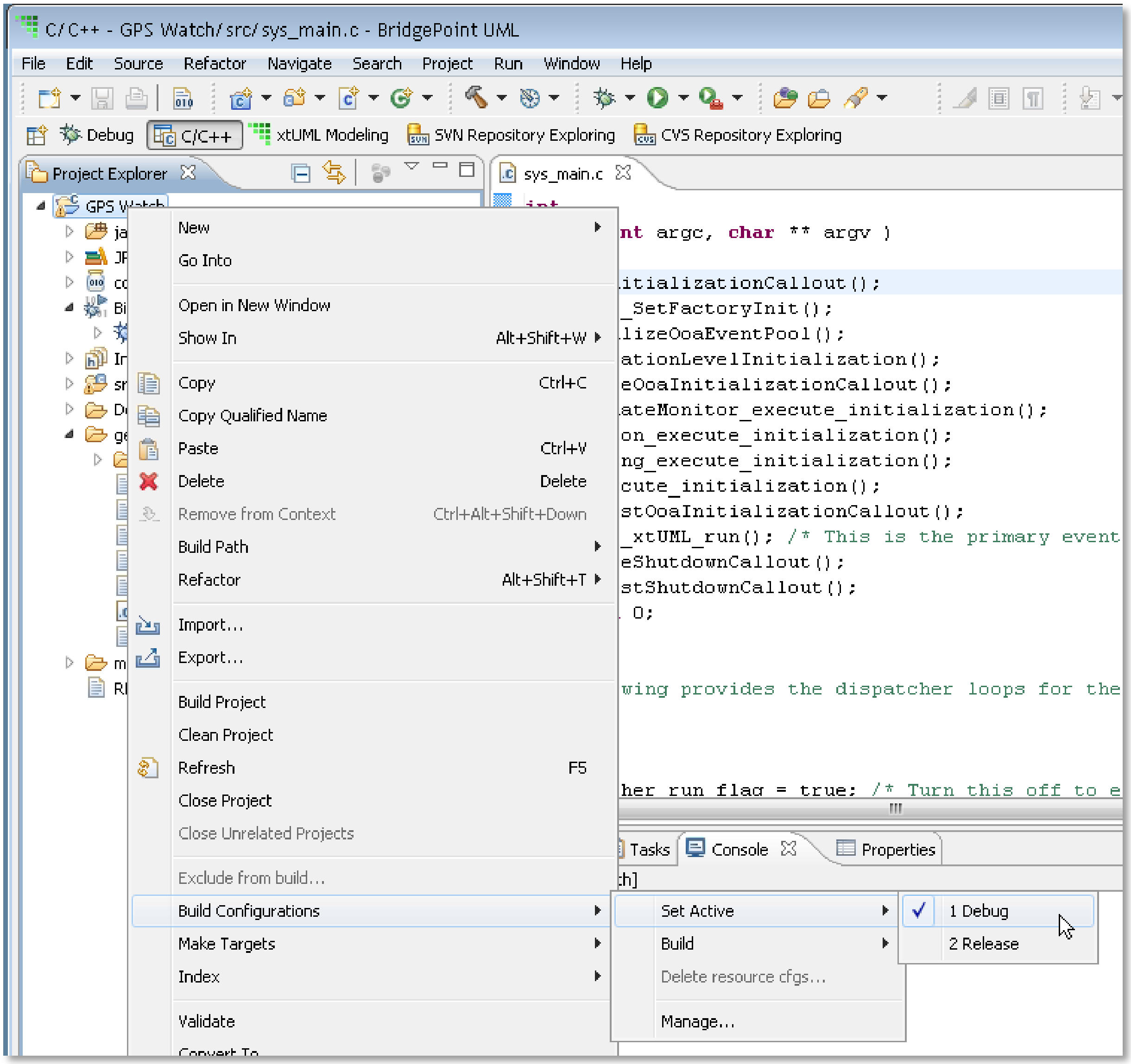
Figure 16

The green bug toolbar icon next to the play button shown in the figure provides a means to launch the application as a debug configuration.

Changing the Build Type

Eclipse CDT sets the default build configuration type to “Debug” in order to facilitate seamless integration with the built-in debugger out of the box. This information comes at a cost, though. It increases the size of the application and reduces execution speed.

At deployment time, switch to the “Release” build configuration to tell the CDT compiler to use settings that will reduce size and increase speed. As shown in Figure 7, the project context menu contains an entry to easily choose the build configuration to use.



```
int
nt argc, char ** argv )
itializationCallout();
_SetFactoryInit();
lizeOoaEventPool();
ationLevelInitialization();
eOoaInitializationCallout();
ateMonitor_execute_initialization();
on_execute_initialization();
ng_execute_initialization();
cute_initialization();
stOoaInitializationCallout();
_xtUML_run(); /* This is the primary event
eShutdownCallout();
stShutdownCallout();
 0;

wing provides the dispatcher loops for the

her run flag = true; /* Turn this off to e
```

Figure 17

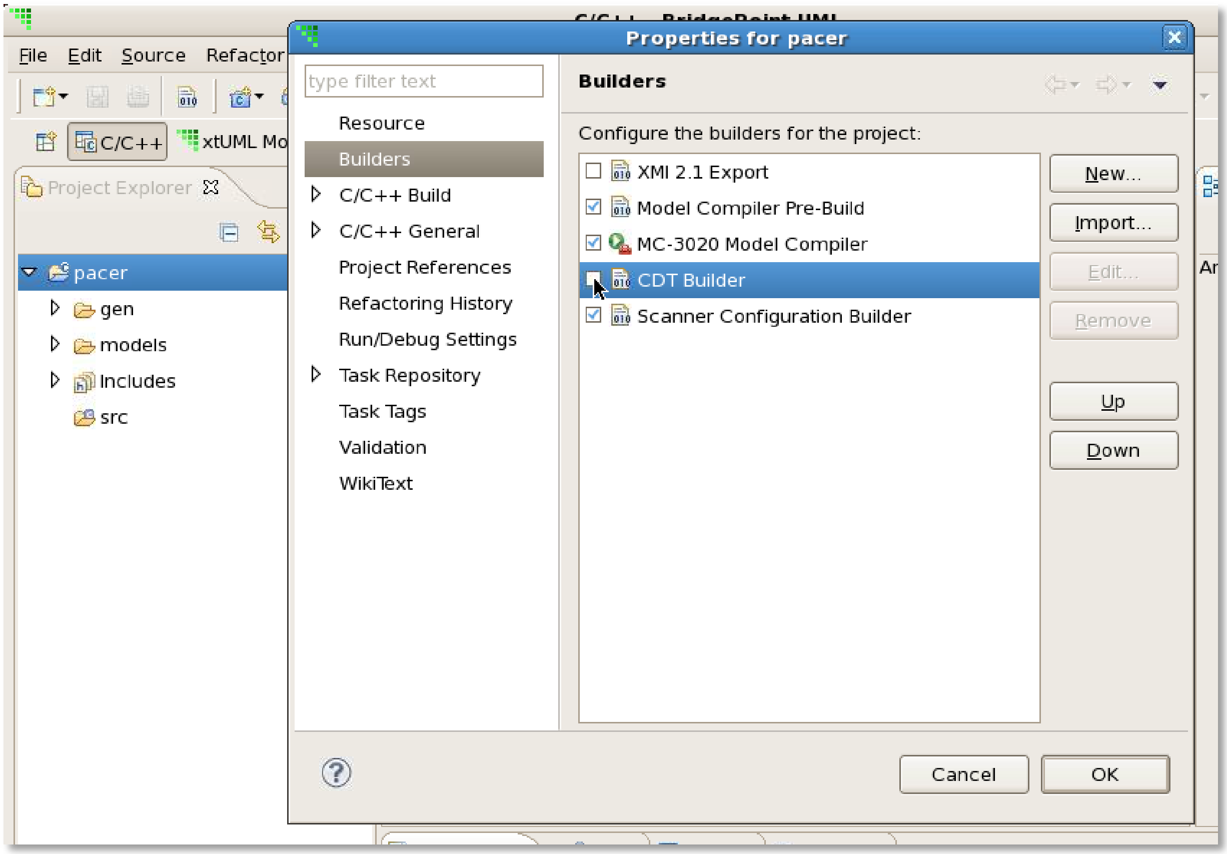
Building and Simulating SystemC-based Models with BridgePoint and Vista

This section describes how to perform the translation of the UML model to SystemC, then import and simulate the generated SystemC code in Vista. This chapter builds on information discussed in the Translation and Implementation Targets sections.

Getting Set Up

The SystemC model compiler is distributed separately from the BridgePoint application installer, see the Configuring BridgePoint section for information about how to install the SystemC model compiler.

The default BridgePoint project configuration enables a C compiler to build the generated source code. Since the SystemC code compilation is performed by Vista, you need to disable the default C code builder in the project properties:



Configure the builders for the project:

- □ XMI 2.1 Export
- ☑ Model Compiler Pre-Build
- ☑ MC-3020 Model Compiler
- □ CDT Builder
- ☑ Scanner Configuration Builder

Figure 18

Marking

BridgePoint uses marking information to provide “knobs and dials” for fine tuning the code compilation process. See the Marking section of the BridgePoint Model Compiler – User’s Guide in the Help system for extensive discussion of this topic. The SystemC-specific Markings subsection provides details and examples for the additional marks you need to know about. For example, the marking information, not the application model is where you choose if you want to create an implementation that uses TLM or non-TLM communication.

You can get started quickly by adding the following marks to `gen/system.mark` and `gen/domain.mark` in your project:

```
system.mark
```

```
.invoke MarkSystemConfigurationPackage( "<your system package>" )
.invoke EnableTasking( "SystemC", "", 1 )
.invoke MarkAllPortsPolymorphic()
.invoke MarkSystemCPortType( "TLM" )
```

```
domain.mark
```

```
.invoke MarkInitializationFunction( "*", "<your init function>" )
```

Building the Project

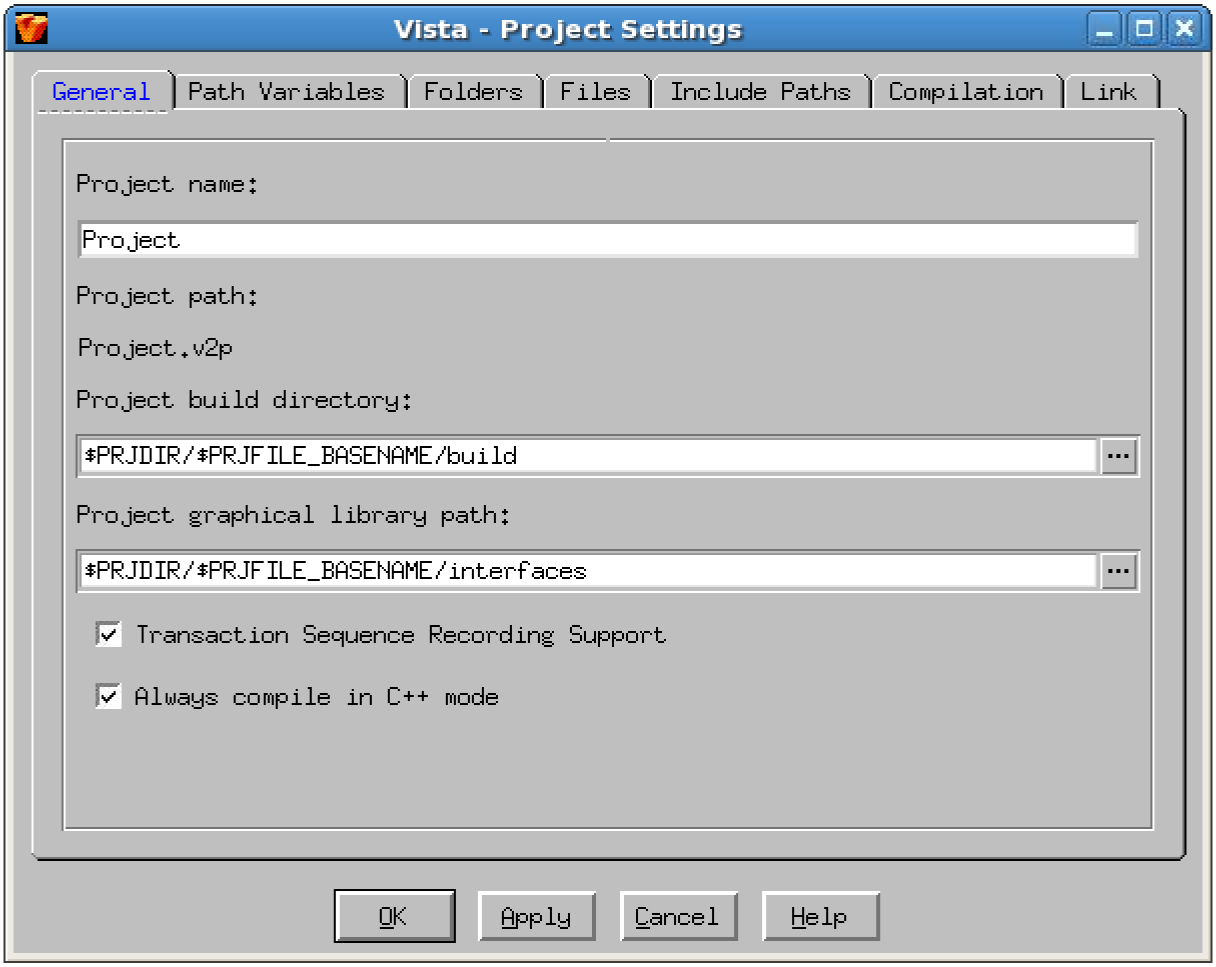
Once the markings are in place, build the project by selecting “Build Project” in the project’s context menu. Informational output from the model translation process is placed in the Console view. The generated code is put into the project’s `src/` folder.

Creating and Configuring a Vista Project

The steps provided here will get the simulation up and running quickly. See the Help contents within Vista for detailed information about the tool functionality.

Once the SystemC source code is created, the next step is to create and configure a project in Vista.

- Make a working directory for vista (e.g. `/home/bpuser/vista/example1`)
- Change to this directory
- Copy the generated `src/` folder from the BridgePoint project into here
- Rename `src/` to `briva_src/`[1]
- Start Vista in the current directory (`$ vista &`)
- Create a project (Project > New Project) named "Project" in the default/current directory



Project name:

Project

Project path:

Project.v2p

Project build directory:

$PRJDIR/$PRJFILE_BASENAME/build

Project graphical library path:

$PRJDIR/$PRJFILE_BASENAME/interfaces

☑ Transaction Sequence Recording Support

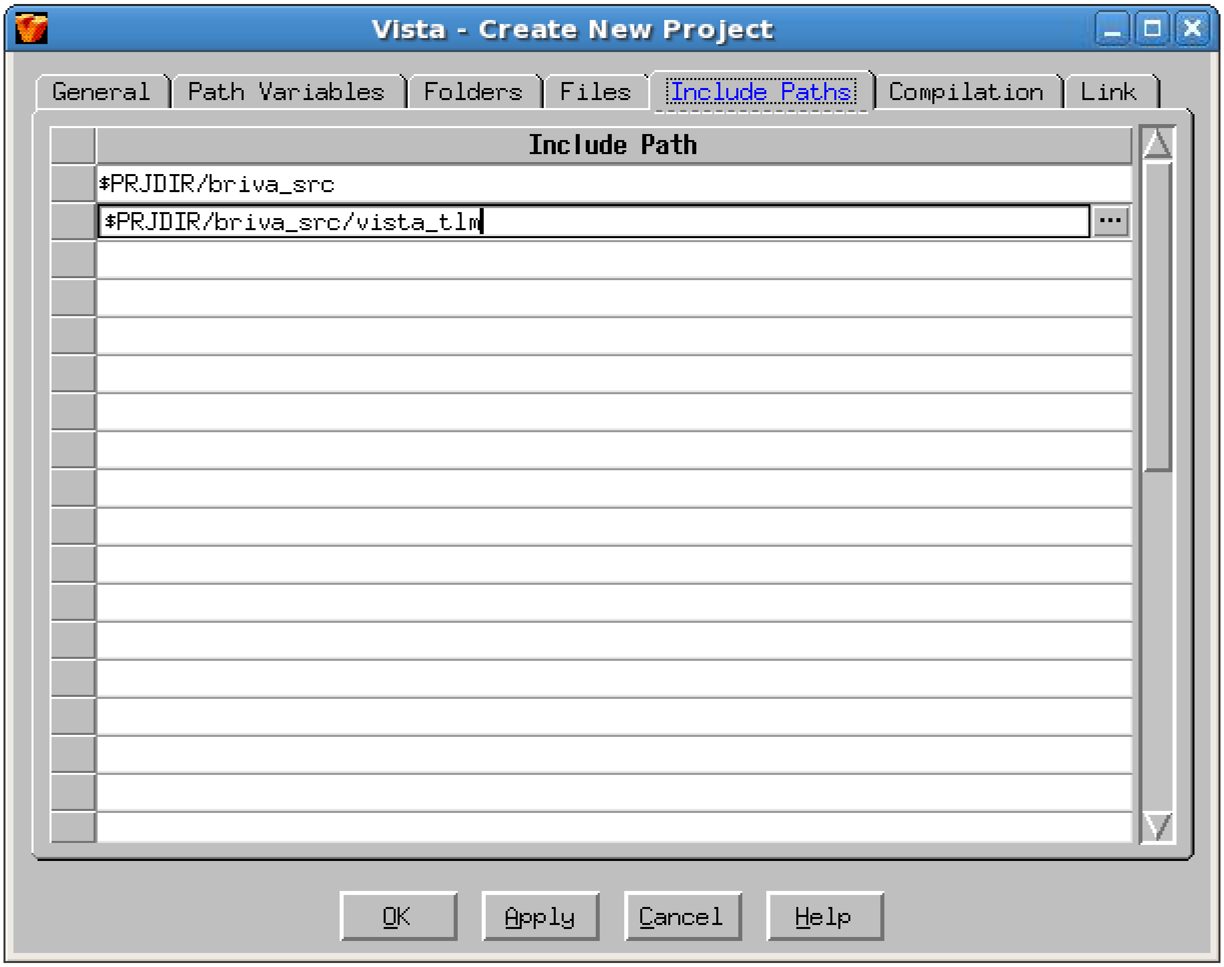
☑ Always compile in C++ mode

Figure 19

- Open the Files tab

- Click "Add Files" button
- Open the "briva_src" folder
- Enter *.cpp in the "File names" field, select "Open"
- Multi-select all the *.cpp files, select "Open"
- Click "Add Files" button
- Enter *.h in the "File names" field, select "Open"
- Multi-select all the *.h files, select "Open"

- Open the Include Paths tab and enter the information shown below:



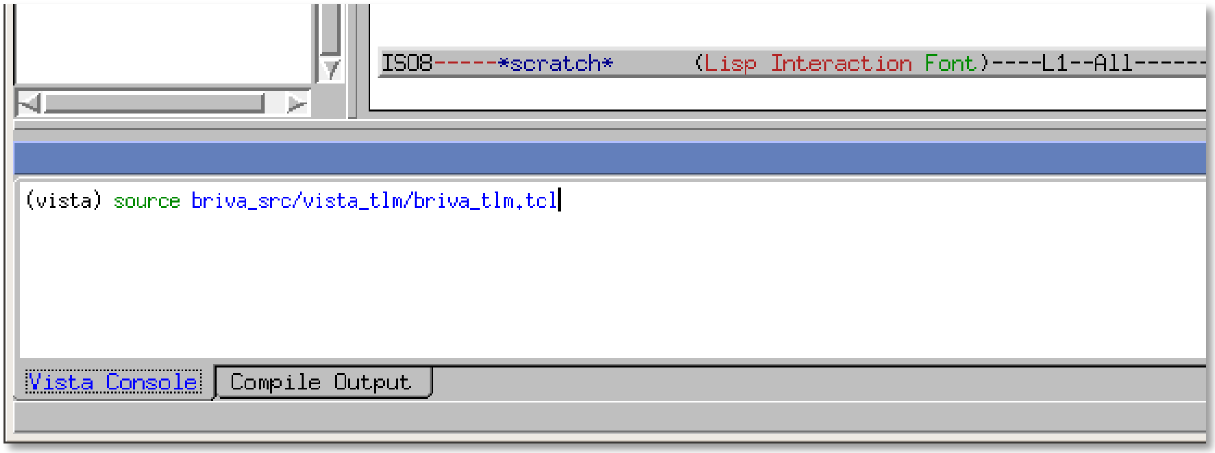
Include Path
$PRJDIR/briva_src
$PRJDIR/briva_src/vista_tlm

Figure 20

- Click OK to close Create New Project wizard

The project is now populated. Before proceeding to the code compilation, create library blocks for the components imported from BridgePoint.

- Click in the Vista Console view at the bottom of the Vista UI and enter the commands:
    - source briva_src/vista_tlm/briva_tlm.tcl
    - briva_all



```
ISO8-----*scratch*        (Lisp Interaction Font)----L1--All------
```

```
(vista) source briva_src/vista_tlm/briva_tlm.tcl
```

Figure 21

The scripts create a library of BridgePoint blocks for use in Vista Model Builder:

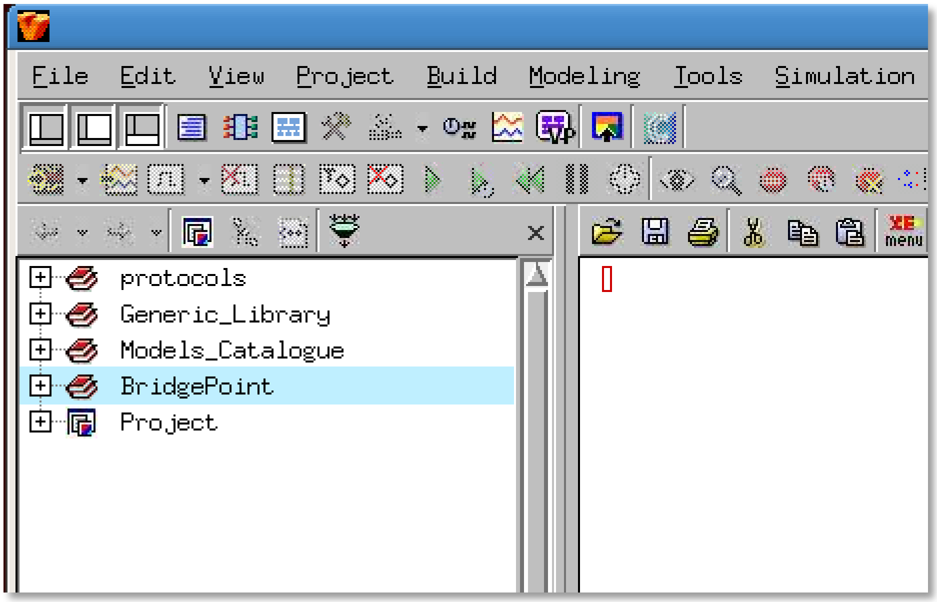


Figure 22

Building the Vista Project

You are now ready to compile the SystemC source code. Simply:

- Highlight “Project” in the tree view
- Build the project (Build > Build Project)

The compiler will output informational messages to the Compile Output view.

Simulating the Project

The compiled code is ready to run in simulation. The generated source code contains an initial testbench and sc_main based on the system package that was specified in BridgePoint. You can also instantiate the elements under the BridgePoint library in other block diagrams for use in the hardware architecture design and analysis. See the Vista help for more information about libraries.

Kick off the simulation using the Project's context menu.

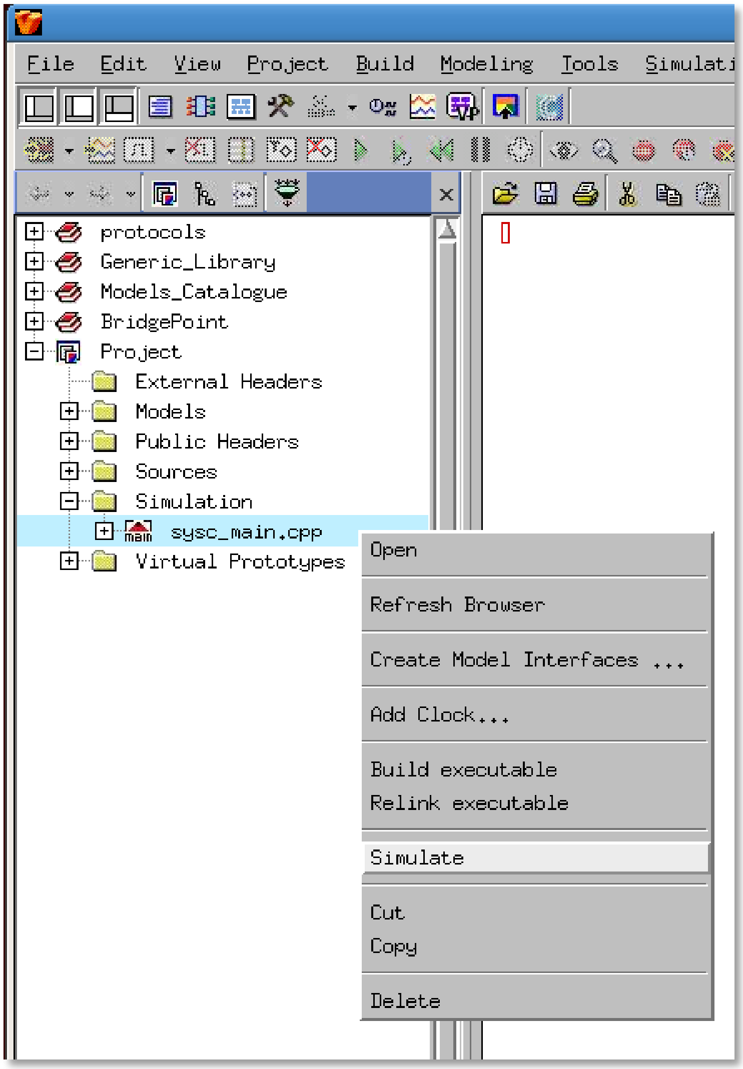
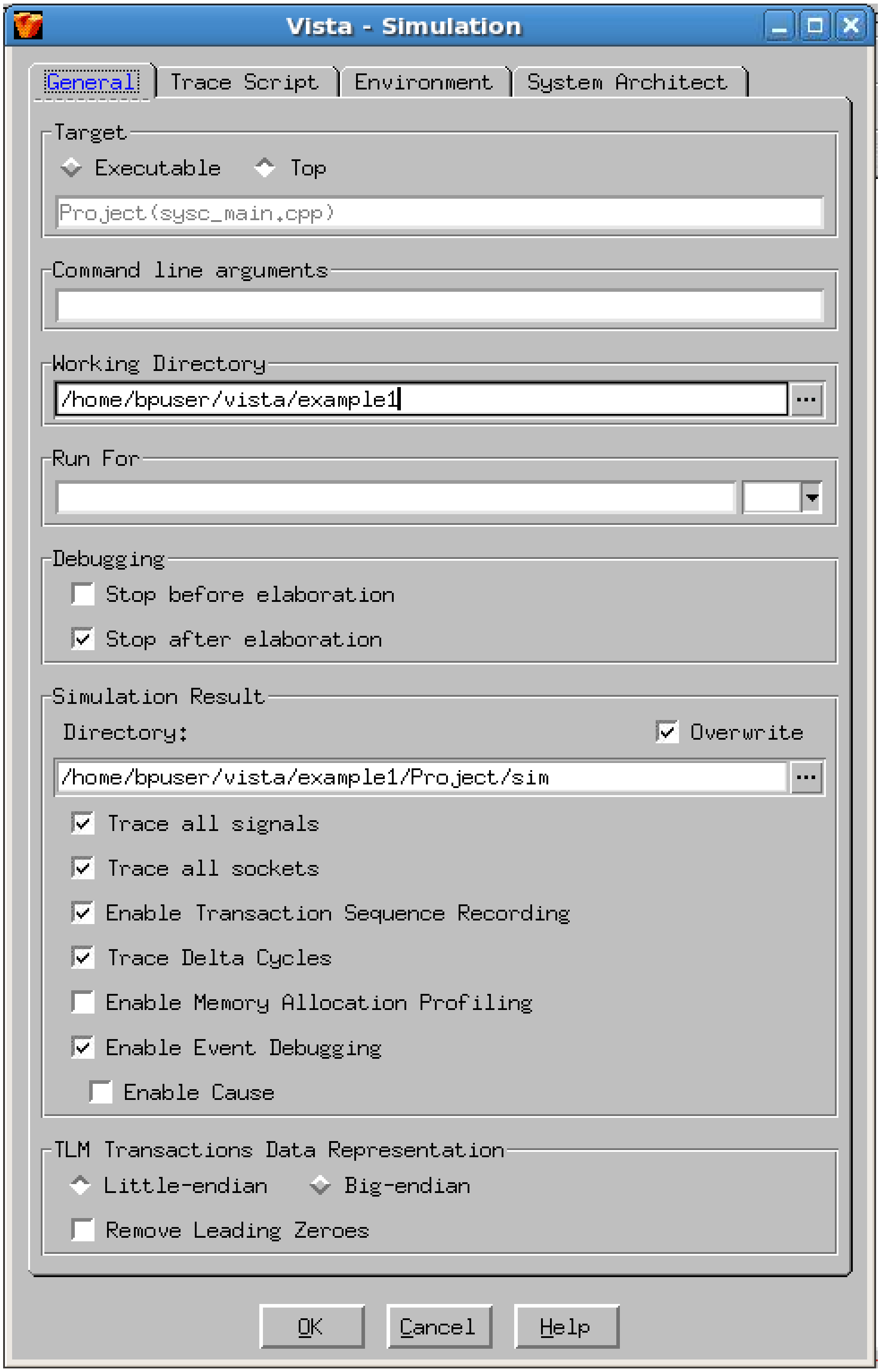


Figure 23

The Simulation wizard starts. Check the appropriate options as shown in the following figure:



Target

◇ Executable ◇ Top

Project(sysc_main.cpp)

Command line arguments

Working Directory

/home/bpuser/vista/example1

Run For

Debugging

☐ Stop before elaboration

☑ Stop after elaboration

Simulation Result

Directory: ☑ Overwrite

/home/bpuser/vista/example1/Project/sim

☑ Trace all signals

☑ Trace all sockets

☑ Enable Transaction Sequence Recording

☑ Trace Delta Cycles

☐ Enable Memory Allocation Profiling

☑ Enable Event Debugging

☐ Enable Cause

TLM Transactions Data Representation

◇ Little-endian ◇ Big-endian

☐ Remove Leading Zeroes

Figure 24

- Click OK. The simulation runs until the top level has been elaborated. After elaboration is complete, Vista shows "Stopped" in the lower right status bar.
- Set breakpoints now (optional)
- In the toolbar, click on the green arrow with a small clock
- Enter a stop time (~5-10 seconds) and then press "Go"
- A lot of output will show up in the Console[2].

[1] If you want to choose a different name in this step, you must modify the `<other src dir name>/vista_tlm/briva_tlm.tcl` file. Update the "bp_source" variable to specify the name you chose for <other src dir name>. All other references to `briva_src/` in the example instructions and screen shots must be modified appropriately as well.

[2] To adjust the size of the console font, click the XEmacs button on the toolbar. Then in the menu that pops up choose "Options > Font Size" and select the desired size.

References

[1] Mellor, Stephen J., and Marc J. Balcer: *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, Boston, 2002. Print. Chapter 18.

[2] Wine URL: *www.winehq.com*

[3] Rumbaugh, James, Ivar Jacobson, and Grady Booch: *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, Boston, 2004. Print. pp111-112.

[4] Rumbaugh, James, Ivar Jacobson, and Grady Booch: *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, Boston, 2004. Print. p304

[5] Mellor, Stephen J., and Marc J. Balcer: *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, Boston, 2002. Print. pp65-67.