The xtUML Modeling Guide

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

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.
- Executable models provide 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, model 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 studying the 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, Stephon 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 in 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 is provided that will enable practitioners to quickly become effective xtUML modelers.

BridgePoint Documentation

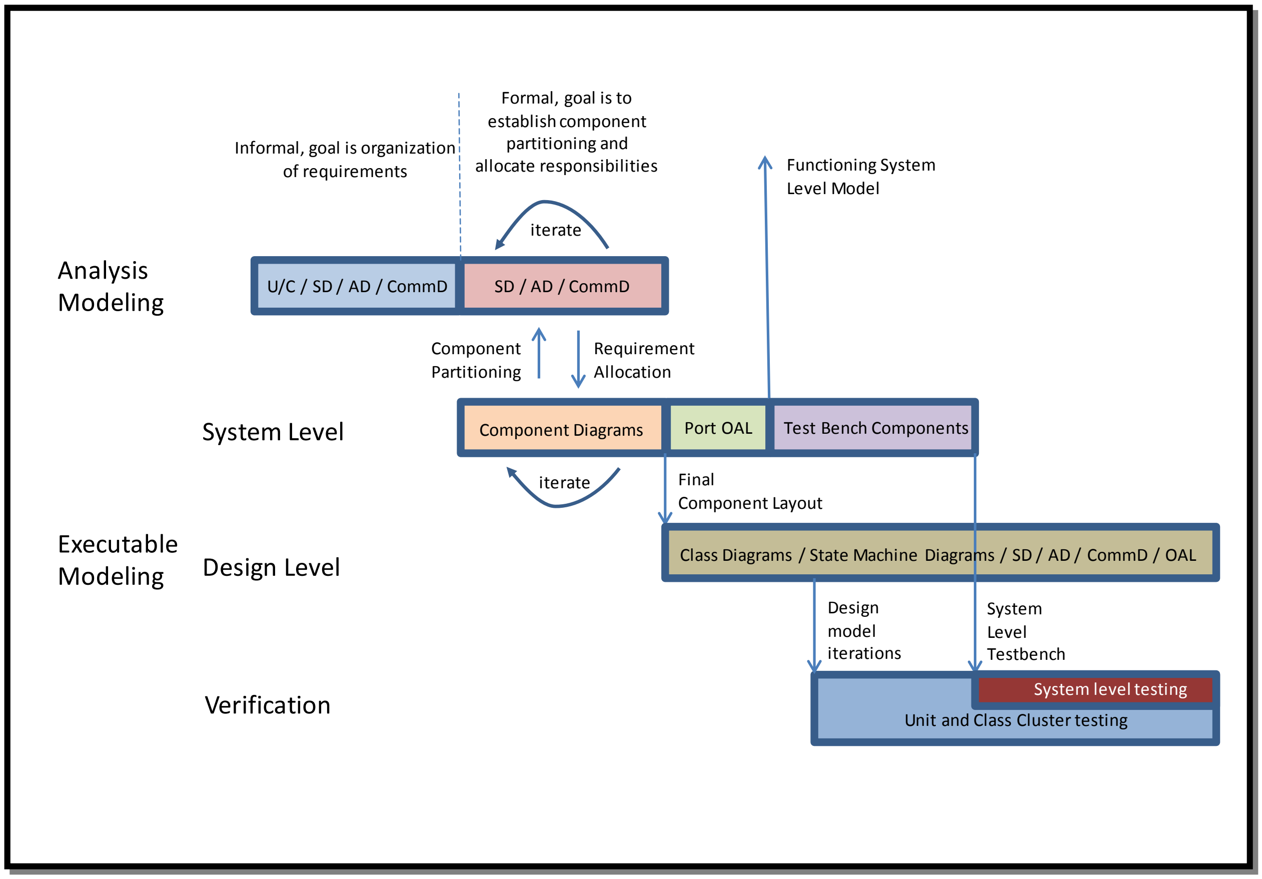
The BridgePoint tool includes reference and user documentation that addresses tool-specific features, details of use of the many features, and examples of key attributes. The reader is referred to this source in addition to the best practices listed in the Style Guide.

Additional Contributions

Other contributions come from the many xtUML users and practitioners across many fields, and their individual and collective expertise is gratefully acknowledged.

Introduction to the xtUML Methodology

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, translation, 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. Each of these phases will be explored in following sections and broken down using a recursive refine technique involving analysis, design, test and review.



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 they 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 offered by 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 formally reviewed and 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 is to be translatable into either hardware or software, then 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 theme will be emphasized throughout the 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. These Markings are tags stored apart from the Executable 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.

Complementary Processes

Documentation of the Design

Well documented models can form a nucleus of design and reference material essential to good design comprehension, and reusable work products. The capabilities to generate documentation automatically are built into the BridgePoint toolset, and can be controlled as to the style, depth and scope of documents created.

This guide also gives recommendations as to the way to best use the various authoring capabilities and diagrams available within BridgePoint to build an effective set of documents that fully describe the design.

Configuration Management

A well-managed design will utilize standard resources to not only capture the design, but also to handle change, increments, braches and variants, etc. The infrastructure of BridgePoint supports and integrates with industry standard configuration management tools, and this guide sets out best practices for the use of Subversion. BridgePoint is also tightly integrated with the Mentor Graphics System Design Management toolset, SDM.

Requirements/Issue Management

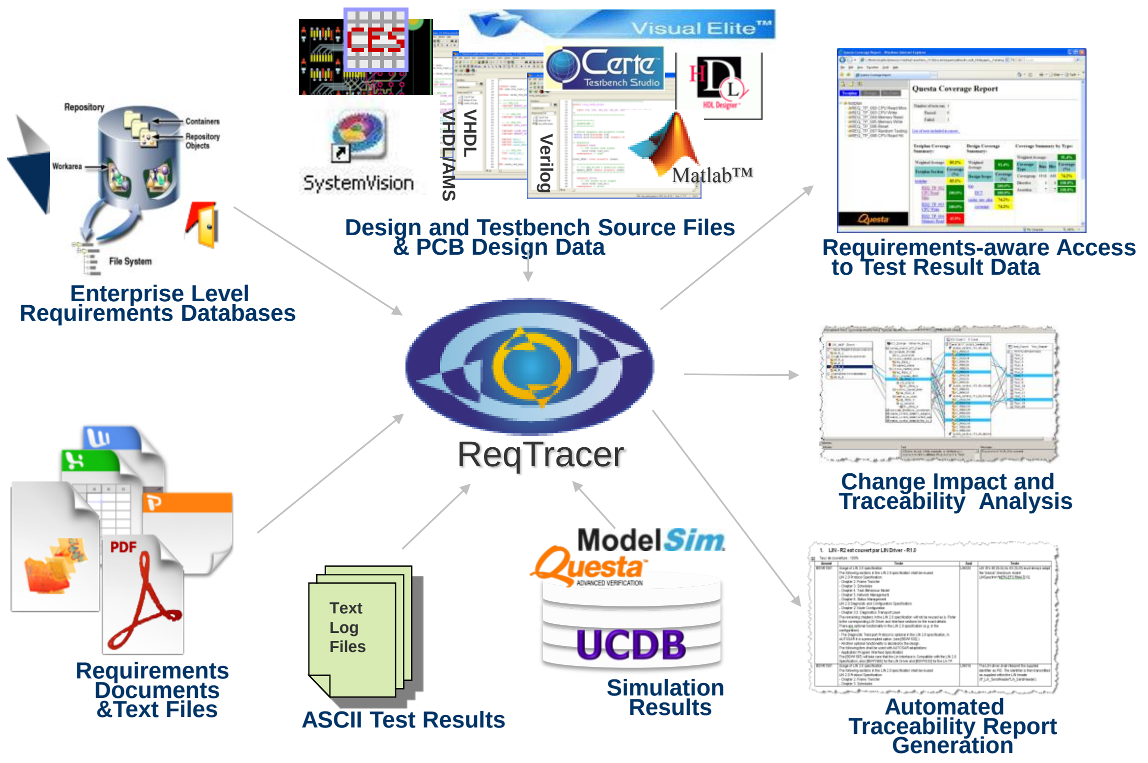
Numerous dedicated solutions are offered for capturing and tracking requirements, issues or changes related to the design. This guide refers to the process of associating requirements, issues and related artifacts with the design, and does not proscribe any particular tools or flows. BridgePoint is also able to support association of these kinds through tight integration with the Mentor Graphics System Design Management toolset, SDM.

The design flow 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 practices, guidelines, processes and flows that meet and lead forward those practices within the enterprise.

Requirements Driven Modeling

Every project starts with a set of requirements that are conditions or capabilities to which the system under construction must satisfy. A good requirement must be Complete, must be Correct and describe what is desired, must be Consistent and not conflict with other requirements, must be Unambiguous, and must be Verifiable. However, the traditional starting point for a project's set of requirements is text, and all too often projects are derailed by relying too extensively on these early documents. The problem with text documents is that they are hard to keep current, difficult to communicate changes, and difficult to links with other project artifacts such as functional requirements, Use cases, Test results and project tasks.

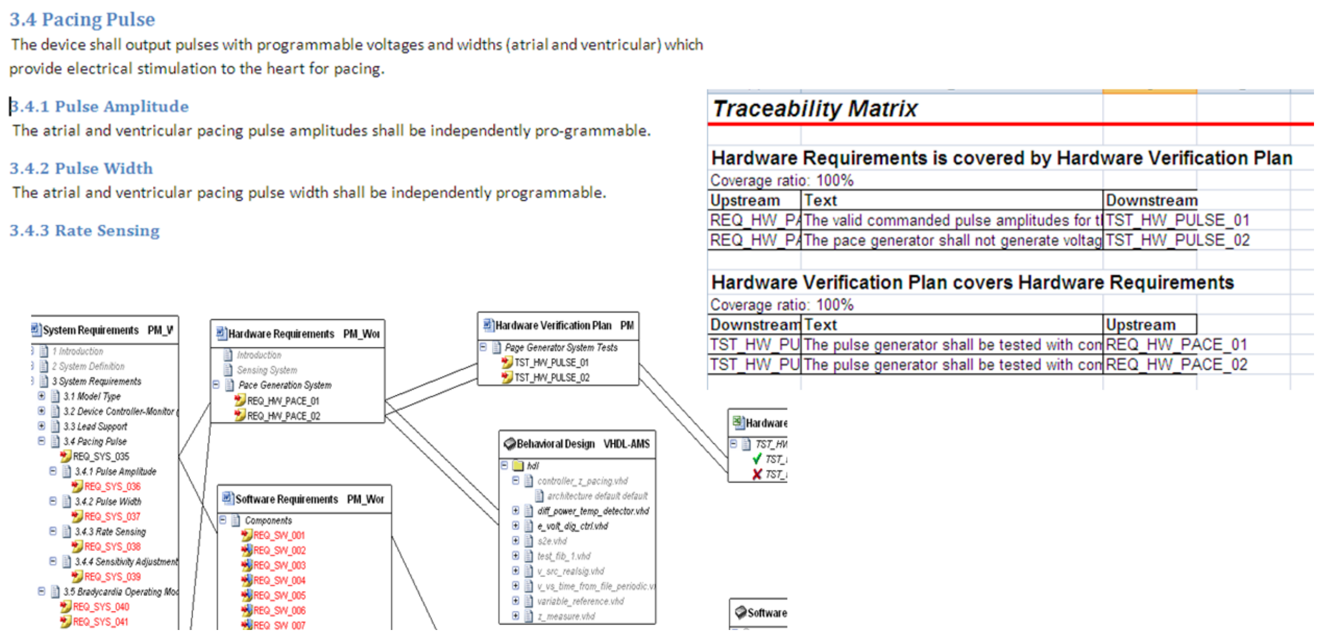

Repository
Containers
Repository Objects
Workarea
File System
Enterprise Level Requirements Databases
CES
Visual Elite™
Certe Testbench Studio
HDL Designer
SystemVision
VHDL
VHDL-AMS
Verilog
Matlab™
Design and Testbench Source Files & PCB Design Data
Questa Coverage Report
Requirements-aware Access to Test Result Data
ReqTracer
Change Impact and Traceability Analysis
PDF
Requirements Documents &Text Files
Text Log Files
ASCII Test Results
ModelSim
Questa
UCDB
Simulation Results
Automated Traceability Report Generation

Since the xtUML Methodology is a Requirements-driven design process, effective teams do not rely exclusively on text documents. Instead, they employ a requirements tool that supports the tagging of design artifacts, the tracing of these tags through out the design process and the generation of summary reports. An example of such a tool is ReqTracer from Mentor Graphics and the capabilities it offers are summarized in the figure above.

The Requirements phase consists of two parts: requirements gathering and requirements synthesis or triage. While requirements gathering can be effectively done using textual artifacts, in the xtUML approach, requirements synthesis employs models as a complementary representation of functional requirements. Models provide a concise unambiguous description of a requirement and reduce the effort involved in requirements bundling. Executable Models go one step further and are verifiable. Consequently, in the xtUML methodology it is feasible to merge the Requirements synthesis phase with early Analysis Modeling.

Next is Requirements triage where groups of raw requirements are identified and prioritized.

xtUML provides a methodology of analysis, design and test that delivers effective Requirements triage by recursively refining the detail, re-balancing the clustering of requirements, and delivering a verified implementation.



3.4 Pacing Pulse

The device shall output pulses with programmable voltages and widths (atrial and ventricular) which provide electrical stimulation to the heart for pacing.

3.4.1 Pulse Amplitude

The atrial and ventricular pacing pulse amplitudes shall be independently pro-grammable.

3.4.2 Pulse Width

The atrial and ventricular pacing pulse width shall be independently programmable.

3.4.3 Rate Sensing

Traceability Matrix

Hardware Requirements is covered by Hardware Verification Plan

Coverage ratio: 100%

Upstream	Text	Downstream
REQ_HW_PA	The valid commanded pulse amplitudes for t	TST_HW_PULSE_01
REQ_HW_PA	The pace generator shall not generate voltag	TST_HW_PULSE_02

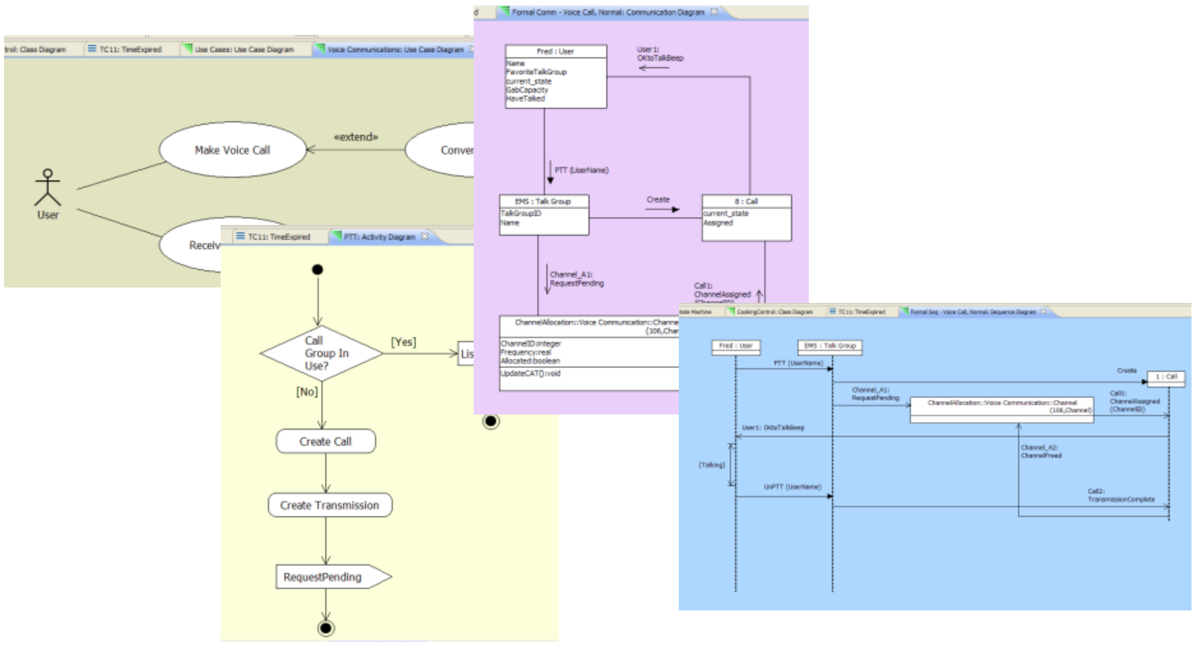
Hardware Verification Plan covers Hardware Requirements

Coverage ratio: 100%

Downstream	Text	Upstream
TST_HW_PU	The pulse generator shall be tested with con	REQ_HW_PACE_01
TST_HW_PU	The pulse generator shall be tested with con	REQ_HW_PACE_02

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. To perform effective Analysis Modeling, a project must have reached the point where Requirements are stabilizing, captured and organized. It is recommended that these requirements be stored in a Requirements tracking repository. Also, textual requirements should be numbered or otherwise supplied with unique identifiers. In general, this phase should not begin until Sign-offs and formal approvals have been obtained on the requirements specification from all Stakeholders.



Steps to performing Effective Analysis Modeling

To perform Analysis Modeling within the xtUML methodology, most effective teams employ the following steps:

1. Build Use Cases, but analyze only system goals and functions down to the level of an achievable single scenario ("kite" level [1,2]). Do not attempt to use these diagrams for documenting detailed algorithms or protocols. Specific instructions on constructing effective Use Case Diagrams is presented at the end of this section.
2. Build Sequence diagrams for requirements that refer to protocols and other temporal features of each requirement. Capture time, sequence, ordering, hand-shaking and other inter-element exchange specifications.
3. Build Activity diagrams for requirements that prescribe detailed algorithms and processing-centric scenarios.
4. Build Communication diagrams to outline patterns of information exchange with direction and ordering of the messages and parameters.
5. 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.
    - Creates, validates and expands the Component diagrams. Note, that this is the initial task in the Executable Modeling phase. It is referred to here, because the two tasks briefly overlap and run concurrently, while provide feedback to each other.
6. Perform early component discovery and capture in a first draft Component diagram. It is not necessary to make these Component candidates executable, yet.
7. Revisit each Use Case, Sequence diagram, Activity diagram and Communication diagram created so far to continue refinement, add detail, and check for correctness and consistency.

8. Referring to the early Component layout, create new more detailed Sequence and Activity diagrams that show how the requirements map onto the candidate Components. Refine and formalize elements from these diagrams to the Components on the emerging Component diagrams. Some of these Sequences and Activities are refinements of ones created earlier. Many will be new, created from mapping Use Case requirements onto the Component structure.
9. Create Interfaces with Messages to support the allocations. Some of the original Sequences and Activities may not be refined at all, because they capture small details of the requirement that are better slotted in later during the main Execution Modeling stage. During this mapping stage, you may find that the original candidate Component structure is sub-optimal. If this is the case, push some Components down into nested Components or into candidate classes. Conversely, other Components may need to be split into two or more new ones.
10. If rework is required then return to step 5 and repeat. Otherwise proceed to developing the Executable Model.

Once iteration is complete, the Sequences and Activities documented will clearly show the partitioning by Component and are ready to feed directly into the Executable Modeling step. In some cases, a Use Case is entirely met by a single Component. In this scenario, the whole Use Case is allocated to the relevant Component, and the Executable Modeling step can begin. The System and Design teams can now independently 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.

The Design teams enter the Executable Modeling phase of the design, armed with a fully organized set of Sequences and Activities that specify the responsibilities of each Component to be developed. When complete, the System team delivers the System level executable work products to the Design teams. They can then use the Test Bench components to exercise the Design artifacts.

Use Case Modeling

Use Case analysis is performed after the requirements are gathered. This analysis results in the creation of diagrams that help to organize and partition the frequently large amount of information gathered during the requirements gathering procedures. Use Cases diagrams are also helpful in the creation of test suites employed during Model Verification.

Use Case Diagrams specify interactions between the system being developed and the outside world. They capture functional behaviors by describing the exchange of information between providers and consumers, often called Actors. Actors need not be humans; they may also represent a sensor or actuator. Conversely, an Actor represents a role, not an individual, so the same human might be represented by more than one Actor, if they interact with the system in multiple ways. Each separate interaction with the system is represented by a single Use Case symbol, named to clearly capture its scope. Actors which are involved in a particular interaction are connected to the Use Case with a 'uses' relationship. Similar Use Cases may be grouped using the 'generalization' relationship, and Use Cases may be broken down into smaller parts (also depicted using Use Case symbols) using the 'include' and 'extend' relationships.

Steps to Constructing Use Case

Once Requirements Gathering is completed and the collected requirements stored in a repository, the early Analysis Modeling phase can begin with the construction of Use Case Diagrams. The following steps are recommended when developing these diagrams:

1. Begin by creating high level Use Cases covering broad system goals. Do not worry about error conditions, yet. This approach, often referred to as 'Sunny Day' or 'Happy Day' scenarios, avoid cluttering the early Use Case diagrams. Use only the terminology found in the requirements repository, avoiding all references to any implementation technology. In the steps below, the "categorize" and "allocate" tasks are performed on paper or on a white board and then transferred to the prescribed diagrams.

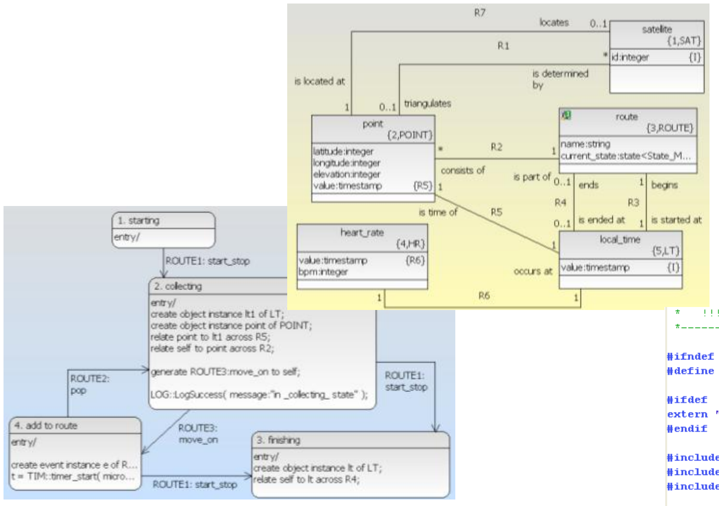
2. Review the requirements repository and identify all functional requirements. Examples of requirements to ignore at this stage are those which refer to performance, usability or reliability goals. Categorize the remaining requirements into four broad levels, "sky" - broad system goals, "kite" – steps required to achieve a goal, "sea" - single atomic interactions and "mud" – a detailed description of some implementation detail.
3. Draw a Use Case Diagram for the "sky" level requirements. Often this will fit on a single sheet. If it does not fit on at most two or three sheets, review the criteria you used for determining what a “sky” level functional requirement is, and repeat this step.
4. Add a Use Case symbol for each "sky" level requirement. Give it a two or three word name, and identify the Actors that must interact with the system to achieve the Use Case goal. Identify unique two or three word names for the Actors. Draw and associate them with the relevant Use Case. Write two or three sentence descriptions for every Use Case and Actor drawn. Review this diagram with the requirements originators. Address observations and iterate, until all parties agree that the diagrams capture the high level goals of the system.
5. Allocate the "kite" level requirements to "sky" level Use Cases. If a "kite" level requirement cannot be allocated to a high level Use Case, look for a missing "sky" level requirement, and iterate with the requirement’s originators.
6. For each Use Case on the top level diagram, review the requirements allocated to it. Create a new Package, named after the top level Use Case, and draw "kite" level Use Case symbols for each allocated functional requirement.
7. Review the "kite" level Use Cases, and identify groups of similar requirements. Document these by adding generalization relationships with the head of the arrow pointing towards the more general Use Case. Similarly, identify Use Cases which represent steps that appear in multiple places, and document this reuse using an "includes" relation. Use the "extends" relation to document Use Cases which are similar, but which add extra steps as necessary. Similarity amongst Actors may also be captured using the Generalization relation. For example, a 'Physician' Actor might be specialized into 'Referring Physician' and 'Cardiologist'.

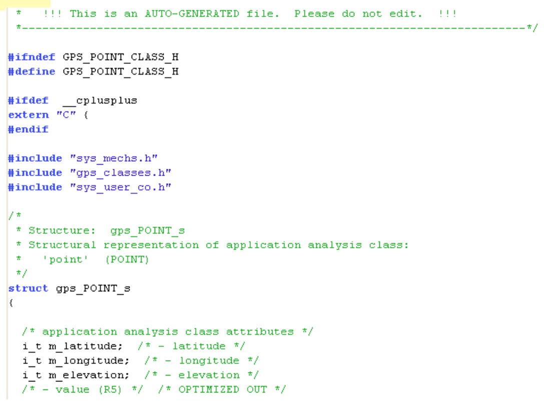
8. Use Cases at the "kite" level are considered to be the best for moving to the next step of the development process. It may sometimes be beneficial to explore "sea" level requirements by iterating the above steps, but this frequently leads to an overwhelming and unmanageable number of diagrams, which do not add value later on. Only for the most complex "kite" level Use Cases will it be beneficial to iterate, and build Use Case diagrams showing the "sea" level Use Cases. Note, it is never helpful to continue to produce Use Case diagrams down into the "mud".
9. For all Use Cases below the top level, produce short statements of pre- and post-conditions. This will provide useful information when it comes time to create test plans, test scenarios and test bench models.
10. Ensure that all Actors, Use Cases and all relations contain full descriptions. Review the models with the requirements team and other stakeholders. Iterate until the Use Cases are accepted and Signed off during a formal review.

The final step in the development of Use Case diagrams is a successful review. Once completed the Analysis Modeling phase can continue, and repeated iterations of refinement performed until sufficient detail is achieved and Exectuable Modeling can begin. The Analysis Modeling discussion includes several recommendations on when to transition from Analysis Modeling to Executable Modeling.

Executable Modeling

Executable Modeling is the task that takes the Use Cases and Scenarios described in the Analysis work products, 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 captured in the form of Use Cases, Sequences, Activities and Communication patterns. The Executable Model takes the form of a collection of components that run, and can be translated into code targeted to various hardware and software platforms.





```
 *   !!! This is an AUTO-GENERATED file.  Please do not edit.  !!!
 *-------------------------------------------------------------------------*/

#ifndef GPS_POINT_CLASS_H
#define GPS_POINT_CLASS_H

#ifdef  __cplusplus
extern "C" {
#endif

#include "sys_mechs.h"
#include "gps_classes.h"
#include "sys_user_co.h"

/*
 * Structure:  gps_POINT_s
 * Structural representation of application analysis class:
 *   'point'  (POINT)
 */
struct gps_POINT_s
{

  /* application analysis class attributes */
  i_t m_latitude;  /* - latitude */
  i_t m_longitude;  /* - longitude */
  i_t m_elevation;  /* - elevation */
  /* - value (R5) */  /* OPTIMIZED OUT */
```

Prior to performing Executable Modeling, the project must have fully completed the Analysis Modeling phase described earlier. From this phase, the following artifacts will be available:

- A decomposition of the requirements into a collection of Use Cases and scenarios that illustrate what happens when the system runs.
- Initial partitioning of the scenarios and mapping of elements into components.

- Communication patterns and ordering of messaging between components documented in the Analysis Models.

Armed with this material, a modeler will follow an iterative process that converges on an Executable Model that fully documents the requirements, is verifiable and is implementable using Model Compilation.

Steps to developing complete Executable Models

To construct an accurate and efficient Executable Model, the following high level steps are recommended:

1. For each Component to be developed, identify all Sequences and Activities that must be supported.
2. For each Component level Sequence and Activity (analysis work product) analyze the analysis work product and if necessary its underlying requirements to identify the conceptual entities ("things") stated or implied by the requirement.
3. Organize the set of things by creating a classification system
4. Create nested Packages under the Component that reflects the classification system.
5. Draw Classes in the appropriate Package for each conceptual entity.
6. Add Attributes to the Classes to capture the characteristics of each entity.
7. Look for patterns of behavior and characteristics that group the Classes based on important features of the requirement. Capture these groupings in Generalization/ Specialization hierarchies using the Supertypes and Subtype Classes.
8. Add Associations to describe the relationships between the Classes, and capture the semantics of the relationship in association end phrases.
9. Capture constraints on those relationships by specifying the cardinality (conditionality and multiplicity) of the Associations.
10. Identify Classes which have a lifecycle and capture it as Instance level State Machines.

11. Optionally illustrate the interactions of the discovered Classes by drawing a new set of Sequence and Activity diagrams that map the Component level analysis work products onto the Classes. These diagrams show how the Classes progress through their lifecycles and interact to meet the requirements.
12. Add operations to the Classes to support the Sequence and Activity diagram construction.
13. For each Operation, State and Transition, write Object Action Language (OAL) that specifies the detailed steps required to carry out the behavior of each atomic activity of the Classes.
14. Review the choice of Classes and partitioning of behavior in a formal review meeting. If necessary, iterate to optimize the design.

System Level Executable Models

System Level Executable Modeling establishes the Component structure of the system being developed. It maps the scenarios identified in the analysis work products onto the Components, and creates an executable system level model that exemplifies the required system behavior. It delivers a library of Component diagrams that show the top level or levels of system decomposition, together with a Test Bench library that drives the system level components through all the required behaviors.

This phase of Executable Modeling begins when early Analysis Modeling is complete and the requirements are organized into:

- Use Cases that illustrate what is required of the system.
- Informal Sequence, Communication and Activity diagrams that show required messaging patterns and algorithmic details.

Steps to constructing Executable Models

At the beginning of this phase we may observe that the Component structure is in one of three states:

- The Component structure does not exist.
- The Component structure is substantially specified. It lacks only the new Components required as a product of the current work.
- The Component structure of the system is already fully specified.

If the component structure does not exist, then all steps should be followed. However, when a candidate structure is borrowed from a legacy model, the initial steps 1 through 4 can be omitted.

1. Begin by examining the requirements and analysis work products to identify the subject matters present.
2. Depending on the starting state, either:-

    - Base the first Component breakdown on the subject matters found.
    - Identify new required Components based on the subject matters found (this activity may well expose extension work required on existing components too).
    - Use the subject matters found to identify the Components which provide a natural home for the new work.

3. Capture the first Component breakdown as a top level Component diagram.
4. If the diagram is too large to be readable on a single sheet, consider nesting components together into a single container component with a larger scope.
5. Now enter an iterative phase that simultaneously:

- Validates and if necessary expands and/or refactors the Component diagrams.
- Begins linking formalized Sequence and Activity diagrams to the emerging Component models. Note: This is a later task in the Analysis Modeling phase. It is referred to here because the two tasks (Component modeling and creation of

formal analysis products) overlap and run concurrently, providing feedback to each other.

6. Work with the Analysis team to allocate requirements and early analysis work products onto the Component structure.
7. Use the identified subject matters to guide the allocation process.
8. As the allocation proceeds, and where the project has control over the relevant work products, add new Interfaces and Messages to support the transfer of required information and events between the components so that the allocation can work. Organize Interfaces in a separate set of Packages because they will be referred to by many different Components.
9. Use the criteria of Coupling and Cohesion to optimize the emerging structure. The early focus on subject matter separation will greatly aid this.
10. Once all requirements and analysis work products are clearly and unambiguously allocated to the given Components, the Analysis phase is complete, and the System Level Executable modeling activity proceeds to the next stage.

Specify the System Behavior in an Executable Form

The second phase of developing an Executable Model focuses on communication between components. For each Interface exposed by each Component under project control, here the steps are:

1. For each incoming message, write Object Action Language in the Component's   Port (Port OAL) that provides a simple required response.
2. The response can be a returned value or a generated signal.
3. For more complex required behavior, such as protocols, create some simple Classes with some Attributes and/or State Machines. Review the informal Sequence and Activity diagrams already created to see if any have captured information that can help with fully specifying these classes.

4. Wire clusters of nested Components together, and test that they interact as expected. Nest tested Component clusters inside container Components by adding references to them in the container Component diagram. Add delegations as required.
5. Proceed by wiring the container Components into clusters until the whole system is represented by an Executable Model.
6. Iterate until each Message of each Interface of each Component has Object Action Language that provides examples of required message passing.

Create Test Benches for Each New Component

The final deliverable in the Executable Modeling phase is a fully verified xtUML model with artifacts that are traceable back to the original requirement. At this stage, it is appropriate to perform unit testing on the model. Note that Model testing is also discussed later in this guide.

To construct a test system model for use in Model testing, perform the following steps:

1. Create a new tree of nested Packages to contain verification Components.
2. For each Component being developed, create a new Package in this tree.
3. Add a reference to the Component to be tested into the Package, and add new Components around it as required to satisfy all the interfaces exposed by the Component under Test (CUT).
4. Now inside each test Component added, write Port OAL that stimulates the CUT through all of its required behaviors. Include Pass/Fail logging.
5. Tests of more complex Component behavior may require some simple Classes, Attributes and perhaps a State Machine to drive them. Detailed recommendations for these complementary test components is provided in the Test section
6. Once the test model performs as required, conduct a review involving analysis, design and test team members.
7. Following a successful review, deliver the test components to the Design team carrying out the Executable Modeling phase.

Identifying Subject Matters

Each subject matter is an independent, self-contained world of concepts. A subject matter is recognizable by the fact that concepts within it refer to each other, but do not depend on concepts in another subject matter. For example, we can extensively discuss a Cardiology subject matter without referring to windows or icons, and we can describe a User Interface without referring to what it is being used for. By identifying subject matters, we focus the attention on what a future Component knows about, as opposed to what it does. Similarly, more attention to what kind of questions will be asked rather than where a particular piece of information is stored in volatile or non-volatile memory.

Steps required for subject matter identification:

1. Review all requirements and analysis work products, and allocate them to a subject matter.
2. If a suitable subject matter does not exist, create it. At the same time as it is created, capture a high level description of the subject matter, and use this to drive the allocation process.
3. If a requirement or analysis work product almost fits within a category, either:

   - Split the requirement while maintaining traceability or,
   - Allocate the requirement or analysis work product to more than one subject matter, clearly recording which parts of it are relevant to which subject matters. Leave no gray areas.
   - Revise the subject matter description to include it. The new description must still satisfy the definition of an independent, self-contained world.

4. Create a root Package to contain the subject matters, and add a Component for each one. Enter the subject matter description into the Component description.

Note that Mellor and Balcer discuss this topic in detail as part of Chapter 3 under the widely used term 'Domain'.

When to STOP

It is very important to decide when the System Modeling activity should stop. The stop point is not a moment in time when all System Engineering team members lay down their mice. Rather, it is specified in terms of a required level of detail for the system level models. We do not provide a single recommendation for this, because project teams vary so widely in composition. Instead, we offer some example project teams, and suggest some completion criteria that could be applied. Every project should look at its human resource profile, and choose an appropriate level of detail to stop at.

Some candidate stopping criteria are:

1. The system team consists of application subject matter experts with little or no software development experience. The design team is qualified software engineers. In this case, consider keeping the range of system model execution paths small. Consider a policy that specifies that Components do not keep any history, and all responses are immediate and based on data contained in the stimulus.
2. The system team contains some software analysts that have a long history with the organization and/or application area. Here, consider allowing a little more detail in the system models. Consider a policy where the system model retains some simple history and responses are based on stimulus plus history. Such history may be captured in simple Classes with perhaps State machines and/or Attributes.
3. There is no separate system team, or most of the system team will migrate to the design team when the Executable System Modeling step is completed. Consider not performing System Level Executable Modeling, or iterating from it to the Design Level Executable Model. If the System Modeling step is omitted, then responsibility for the creation of Test Bench Components falls to the Design level modeling team. Where this happens, a 'test first' approach (some time called Test Driven Development) is recommended, and test artifacts should be specified early in the Executable Modeling phase.

So, the amount of detail to be included in the System Executable Model can vary from zero to one hundred percent of the requirements depending on team makeup. In spite of this wide spectrum, it is extremely important to specify the required level of detail. We do not want two teams simultaneously doing the same things. The goal of the system level model is clarification of requirements, while the goal of the design level model is to specify a solution.

Where there are two teams, the systems team takes responsibility for the Component boundaries and Interface definitions for the duration of the project. Thus, when Component level changes are required, a system team member is always involved in reviewing and approving the change.

Creating loosely Coupled Components

Coupling is an informal measure of the dependency of Components on each other. Loosely coupled Components are easier to maintain and more reusable. It is not possible to specify an integrated system without some coupling. However, some types of coupling lead to less maintainable systems, and should be avoided.

The types of coupling are, in order of desirability:

Data Coupling	Loose
Stamp Coupling	
Control Coupling	
Common Coupling	
Content Coupling	Tight

Data Coupling

Interaction is achieved by passing a small set of arguments in a message. No parameters are redundant.

Stamp Coupling

Interaction is achieved through passing structured data arguments in a message. Some of the members of the type are ignored by the recipient.

Control Coupling

Messages carry flags that tell another Component how to behave.

Common Coupling

Two Components communicate by setting data elements in a third. Design using Components makes this more difficult for a good reason, but it is still possible if you work at it. The recommendation is, don't work at it.

Content Coupling

One Component directly writes data in another. Effectively impossible with Component based design and included here only for completeness.

Creating Components with high Cohesion

Cohesion is a term that relates to the strength of subject matter separation achieved in the Component structure. The types of cohesion, again in order of desirability:

Functional Cohesion	High
Informational Cohesion	
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Low

Functional Cohesion

All allocated responsibilities refer to or manipulate characteristics of the same well bounded subject matter area.

Informational Cohesion

Responsibilities operate on the same data structures, but the data itself is hidden within the Component.

Sequential Cohesion

Responsibilities are grouped together because they refer to the same data structures in some order, such as a pipeline.

Communicational Cohesion

Responsibilities are allocated together because they operate on the same data structures.

Procedural Cohesion

Responsibilities are grouped because they occur one after another, without regard to whether they are referring to the same subject matter.

Temporal Cohesion

Responsibilities are grouped based on when they occur. Component is often named after the significant time, such as 'Initialization'.

Logical Cohesion

Responsibilities are accessed entirely by outside Components. Often, the behavior is accessed through a small interface (one with few defined messages) and arguments specify which behaviors are triggered.

Coincidental Cohesion

Component responsibilities are unrelated. It is often difficult to find a name for the Component, other than 'Miscellaneous' or 'Utilities'.

Creating Test Bench Components

These are not deliverable to end customers. They are initially used to assure that the system requirement is complete. Later, they are used to assure that the design satisfies the requirements. Finally, as the project moves into the maintenance phase, they provide a source of high level regression tests to keep the product on track as variants are produced with enhancements or extensions.

Test Bench Components are created using standard criteria, such as functional testing, code coverage, and boundary condition analysis.

Executable Class Modeling

This phase is performed after the Analysis and Component modeling phases are complete. The goal of Executable Class Modeling is to create formally Executable Models of the requirements that can be tested using Verifier, and transformed into executable code into the selected architecture. Class diagrams are used to capture the detailed patterns of characteristics and behaviors that are present in or inferred from the requirements.

The Executable Modeling phase proceeds by capturing patterns of characteristics and behavior present in the analysis work products. These patterns are partitioned by the Components of the system. The characteristics are captured in units called Classes. Once the Classes have been identified and modeled, the behavior required of each Class will be formally captured in them using Object Action Language (OAL).

Note to Class modelers working in methodologies emphasizing functional decomposition:

> Class modeling focuses on data structure and functional behavior equally. Historically and for good reasons, implementation-centric flows have put much more emphasis on the functional aspects of the design. While you can certainly use BridgePoint to document behavior without exploring the structure of the data very deeply, it is worth considering that modern target hardware provides increasingly large amounts of Random Access Memory (RAM) as part of the intrinsic resources. More RAM means more data to manage, and modelers of data intensive applications have long found that the Class concept is superior at binding the data, state and functional aspects of a design seamlessly and reliably. We encourage you to explore these concepts fully before dismissing their benefits.

Things to know:

1. The requirements repository must be complete.
2. The Analysis phase work products will be complete or at least substantially stable.
3. The Component model must be complete and you must know which Component you will be developing and which test bench Components you will be interacting with.

Class Identification

Start by searching the analysis documentation. All analysis deliverables are relevant; Use Cases, Sequences and Activity diagrams. In an ideal world, every aspect of the requirement will be documented in these diagrams, but do not be afraid to refer back to the requirement repository or the original textual requirement specification if you suspect something has been omitted or incorrectly transformed in some way. You can take each analysis work product one by one to partition the work.

Look for the nouns in the analysis work products that are describing the "things" that the required system must capture and manipulate information about. It is helpful to consider the following, broad categories of "things" during the search:

- Tangible things
- Roles played by people or organizations
- Incidents; an occurrence or event
- Interactions; something which binds two or more things together
- Specifications which prescribe recipes or configurations

As for the analysis procedures, limit your list of candidate "things" to those found in the requirement and stated in the vocabulary of the problem space. This is not always easy, but keep in mind that there will be time to add in technical details later, and Class Diagrams are not always the best way to capture them in any case.

Review the list of "things" you have identified. Write a short description of each thing. If you cannot describe the concept you have captured in a few sentences then consider the possibility that your candidate "thing" is possibly a group of concepts. If this is the case, add your new concepts to the list, and use them to replace the original one, or refine the original definition to distinguish it from the new concepts. Identify characteristics associated with each candidate thing. A characteristic is some atomic piece of information associated with that thing and must be tracked by the system.

Review this list of candidate things with your peers, challenge each entry, and reach a consensus that the list is complete. Complete means that it contains every concept required to support the requirement being satisfied. Complete also means that every "thing" in the list has had its place justified. Be suspicious of any concept that has neither characteristics, nor identifiable interactions or relationship with other abstracted concepts.

Decide on a classification system that will allow you to find items in your collection of things quickly and easily. Use this classification to create a tree of nested Packages named after the levels in your classification. To help with the classification look for clusters of concepts, which are tied together by Incident or Interaction "things". In these clusters, look for a central concept, that if it didn't exist then most or all of the surrounding things would not be needed either. Name the Package after this central concept, this package will contain the Classes created for this group of concepts in the next few steps. This task is really no different than creating a paper filing system. There is no right or wrong answers, but some naming systems are better than others, so do take some time thinking about how you want to organize things. If it makes it easy to find what you are looking for, then it’s good.

Now create classes for each thing in the list you have created.

If this is not the first diagram being created, then review the list of Classes already created. Next, identify those concepts that have already been captured. If a captured Class is similar, but not exactly the same as the one you are about to capture, work with the creator of the existing Class to harmonize the descriptions and characteristics. If the new description is too long, contains statements that are not applicable to some scenarios, or if the list of characteristics is not applicable in all cases, consider creating sub-types of the duplicate that obey these rules (see below). Migrate attributes to the sub-types as required, and adjust all descriptions as necessary.

Open each leaf Package and create a Class symbol for each "thing" you have identified in the cluster identified above. It is common that one of the classes in the Package has the same name as the Package itself. If a concept is marked as already created, create an Imported Class, and assign it to the existing Class instead. For new Classes, capture your descriptions into the Class. Add the characteristics you have found as Attributes of the class. Each characteristic should have an obvious simple type, such as an enumeration, integer, real or string. Never accept string Attributes that specify a format in the description, break these up into separate Attributes or a Structured Data type. Add a description for each Attribute too. Adding descriptions for everything may seem like unnecessary work, but the names you create for the classes and attributes are rarely unambiguous, and may not mean exactly the same to everyone on the team. Creating descriptions avoids the confusion and assumptions that remain the enemy of a smooth development process.

Now start to document the relationships between different classes. Identify groups of like classes, and document this using the Supertype/Subtype association. Where one concept has an identifiable numeric relationship with another (such as, 'a car has four wheels') capture this by drawing an association between the classes. Capture the numeric relationship by categorizing it as one-to-one or one-to-many, and as mandatory or optional. For each association, identify phrases that make the association readable in a way that documents what you have discovered about the requirement that motivated you to add the association. Devise phrases so that they read well no matter which Class you started reading at. Write a full description for each association added.

When complete, review the diagram with your peers.

Specifying Class Behavior

In order to be executable, we will need to add more detail to formally define the required behavior. We capture behavior by adding States, Transitions and Operations to the classes and writing Object Action Language (OAL) for them that specifies what needs to happen when a given stimulus occurs. A description of the structure of OAL is outside the scope of this document, and interested readers are directed to The OAL Reference Manual.

Start this work by constructing Sequence, Communication or Activity diagrams for each "sea" level requirement. The procedure for creating each of these diagrams is found in the discussion of Analysis Modeling and early Executable Modeling.

Search the list of interfaces attached to the Component container to identify the incoming message that initiates the Use Case, Sequence or Activity being worked on. It is possible that the analysis work product is actually addressed across multiple components, in which case there will be a component level Sequence or Communication diagram that illustrates the portion of the Use Case to be addressed by the component being worked on. When constructing formal Class models, the Sequence or Activity diagram must also be created using formal procedures. Following the procedures for the relevant interaction diagram, work through the required behavior, adding Operations and Attributes to the existing Classes to support the interaction being documented.

It may be necessary to create additional helper classes to progress the interaction documentation work. These classes often fall into two categories; Control and Presentation. Presentation classes are often put between the Actor and the core Class model (the classes derived from the requirement subject matter). Presentation classes guide the user through the steps needed to achieve a goal as documented in a Use Case. They should be organized in a separate Package hierarchy since they form the basis for developing the User Interface. Control type classes organize the configuration or reconfiguration of the core Class model instances to satisfy the requirement. They are added to the same Package hierarchy as the core model.

When a need for additional Classes is identified, do not add them directly to the Sequence or Communication diagram. Instead, add them to the appropriate Class diagram Package, then create and formalize a participant to represent them on the Sequence diagram. Similarly, where a Sequence diagram calls for a message to be handled by an existing class, add the operation or event to the Class on the Class diagram, and then reference it from the Sequence diagram. This saves time by not requiring the conversion of an informal class feature to a formal one. The same advice applies to the addition of Attributes.

Review the created Sequence diagrams and when all outstanding observations are addressed, add state models and states to each class as required to fully support any state based Sequence Diagram behavior. Creating state-charts is covered in detail in the next section. Similarly, Activity diagrams may be created to document non state based behavior. Once again, review these diagrams and address observations.

Now, use the Sequence and Activity diagrams to guide construction of the Object Action Language for each State, Transition and Operation. Review all the new material.

Once all OAL is complete, build a test bench model that provides stimuli to drive the model through all of its use cases. Use Verifier to execute the test bench. Confirm that the expected behavior is documented in the Verifier execution transcript and deposited into the test artifacts repository for future reference.

State Modeling

State modeling is performed after the Classes have been abstracted. They confirm the correctness of the class abstraction process and document the behavior of the Classes. State Machines are used to formally capture the lifecycle stages and incidents that an instance (or occasionally a class) may encounter. Instances of a Class generally have a recognizable lifecycle, and occasionally the Class itself may have its own distinct lifecycle. A lifecycle is formally captured as a State Machine. State machines comprise; States which are occupied for significant periods of time (significant may be measured in hours, seconds or microseconds depending on the subject matter), Events which occur at a moment in time, and Transitions which specify which new State will be occupied when an Event occurs in a given current State.

State Machines are part of the formal statement of a systems behavior. Of course, it is possible to create a State Machine diagram that is not intended to be executed, but the steps required to obtain the best model are the same.

First, we examine the requirements we want instances of our class to satisfy. If there is a recognizable set of steps or stages that the instance goes through, then we create an instance State Machine diagram for the Class. List the stages and give each stage a good unambiguous name.

For each stage you listed, create a State symbol and give it the name you selected. Now look for significant incidents that affect an instance of the Class. Enumerate these, again taking the time to give each incident a good clear name. Add a new Event for each identified incident, and give it the name you selected.

Now, for each State on the diagram, consider each Event in turn and decide whether it will cause the instance to progress to a new stage. If it does, decide what State the instance will occupy after the Event occurs, and add a Transition from the starting State to the new State. Write the name of the Event next to the Transition.

After adding a Transition, it is essential to return to the list of Events and continue considering them for applicability in the source State. This is because a given State may have more than one Transition exit path, and it is important to capture them all. When examination of the Event list is complete, move to the next State and repeat the procedure.

When all States have been visited, review the State/Event matrix (States down the side, Events along the top). Fill in the matrix cells for the transitions already found by writing in the name of the new State. For each empty cell (that is, each combination of State and Event) that is empty, consider these three possibilities:

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

In the first case, mark the cell CH. In the second case, enter EI. In the third case, identify the State that the Event should have caused a transition to and write its name in the cell. Review all Matrix cells in this way; do not be tempted to skip any cells.

For each State, consider whether an instance can begin its life in that state. If so, consider whether instances need to be created asynchronously (that is, by sending a Creation Event to the Class). If this is the case, show a Creation Transition flowing into the required starting states. Multiple Creation Transitions are allowed so long as each Event/State combination is unique.

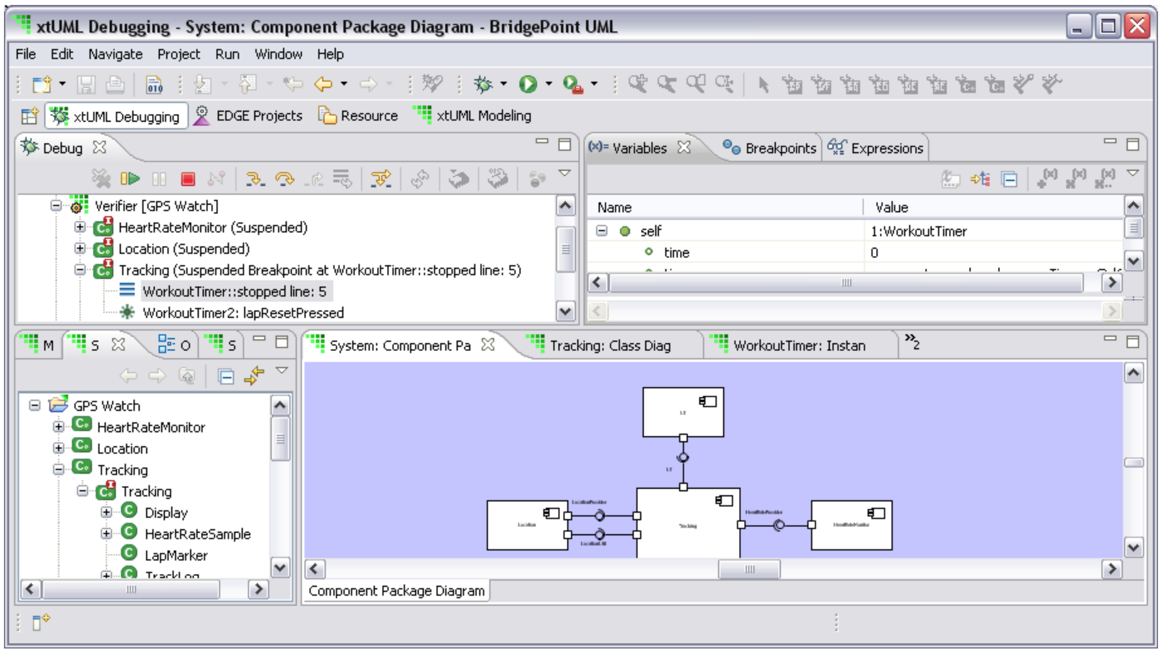
To create a class based State Machine, the steps are the same as for an instance based State Machine. The difference is in the preparation of the lists of States and Events. The States and Events must be common and apply equally to all instances of the class. In other words, an arriving Event causes a State change for every instance simultaneously. Alternatively, the class being worked on may be designed to not have any instances at all.

Secondly, examine the list of Interfaces exposed by the containing Component. For each Interface, look at the incoming Signals and consider whether any are relevant to the life cycle of the class being worked on. If so, add them to a third list.

Now perform the same steps described above, but using these class based lists of States, Signals and Events. For all Transitions, consider Signals as well as Events.

Model Verification

Model Verification is the step that confirms the Executable Model satisfies the requirements captured in the Requirements repository and in the Analysis Model. The steps to achieving effective Model Verification mirror those discussed in the Executable Model section. To perform effective Model Verification, a project must have reached the point where Requirements are stabilizing, captured and organized. It is preferable that these be located in a requirements tracking repository). Textual requirements should be numbered or otherwise supplied with unique identifiers. Sign-offs and approvals have been obtained on a requirements specification.


xtUML Debugging - System: Component Package Diagram - BridgePoint UML
File Edit Navigate Project Run Window Help
xtUML Debugging
EDGE Projects
Resource
xtUML Modeling
Debug
Verifier [GPS Watch]
HeartRateMonitor (Suspended)
Location (Suspended)
Tracking (Suspended Breakpoint at WorkoutTimer::stopped line: 5)
WorkoutTimer::stopped line: 5
WorkoutTimer2: lapResetPressed
Variables
Breakpoints
Expressions
M
S
O
S
GPS Watch
HeartRateMonitor
Location
Tracking
Tracking
Display
HeartRateSample
LapMarker
TrackLog
System: Component Pa
Tracking: Class Diag
WorkoutTimer: Instan
Component Package Diagram

Name	Value
self	1:WorkoutTimer
time	0

Steps to performing Effective Model Verification

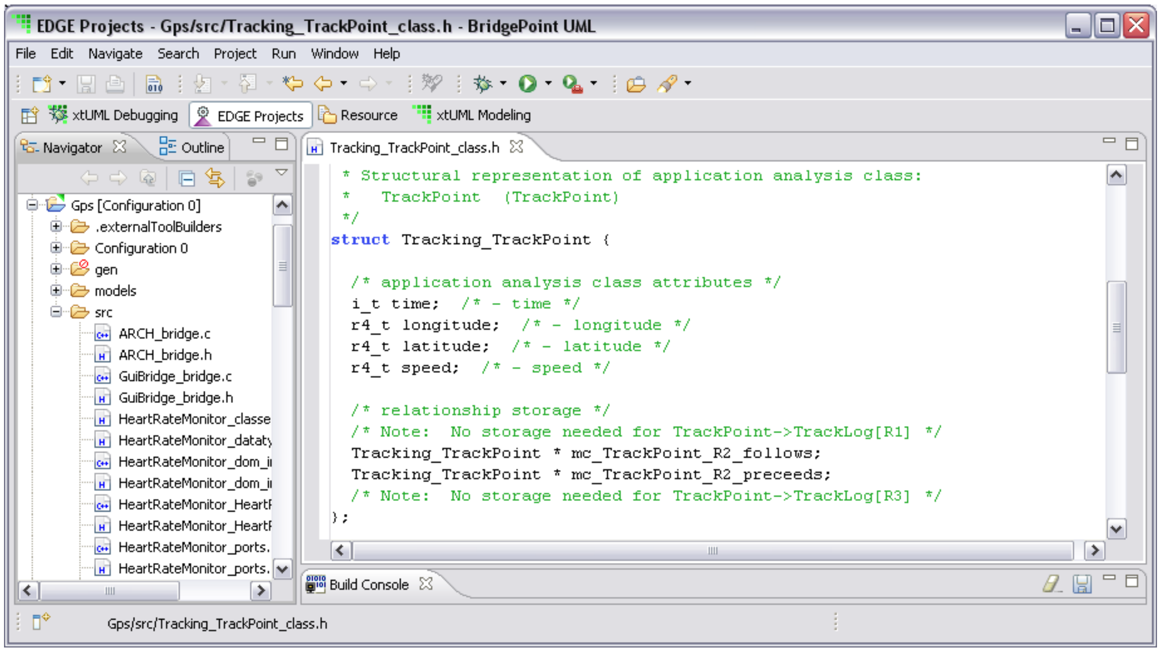
To perform accurate and efficient Model Verification, the following high level steps are recommended:

1. Create test scenarios to drive each class though 100% of its control paths and lifecycle. These test scenarios are modeled as internal "unit test" functions and drive stimulus events and operations to the classes and test for expected post conditions. It may be necessary to write Object Action Language (OAL) that establishes initial conditions for these tests.
2. Execute these scenarios in Verifier. Build up from single classes, to larger and larger class clusters. Ultimately

Use the Test Bench Components supplied by the System level engineering team to exercise and validate the entire Component in Verifier

Model Compilation

Model Compilation is the final step that moves the project from an abstract model representation into a specific Implementation. The key technology used in this step is the model compiler which takes the formalized diagrams and action language blocks of the model, and combines them with a set of compilation rules to produce an implementation source that can be used in the target. Prior to performing this step, the project team must have completed design and verification of the Executable Model. Completion of these steps should be signaled by the process artifacts and the formal review signoff.



```
 * Structural representation of application analysis class:
 *   TrackPoint  (TrackPoint)
 */
struct Tracking_TrackPoint {

  /* application analysis class attributes */
  i_t time;  /* - time */
  r4_t longitude;  /* - longitude */
  r4_t latitude;  /* - latitude */
  r4_t speed;  /* - speed */

  /* relationship storage */
  /* Note:  No storage needed for TrackPoint->TrackLog[R1] */
  Tracking_TrackPoint * mc_TrackPoint_R2_follows;
  Tracking_TrackPoint * mc_TrackPoint_R2_preceeds;
  /* Note:  No storage needed for TrackPoint->TrackLog[R3] */
};
```

Steps to performing Effective Model Compilation

The steps to run the BridgePoint model compiler on a project are:

1. Change to the C/C++ Eclipse perspective.
2. Right click on the project and select "Build Project".
3. This will initiate model translation resulting in the generation of target source code being emitted into the /src folder of the project.

4. The C Development Toolkit (CDT) will automatically cause compilation of the generated source code to be initiated.
5. Check the Console view for errors in the code generation or compilation phases.
6. Find compiled executables in the project folder hierarchy after code compilation has completed.

Model Compilers provide a means for users to select among options and configuration settings for the code generation. This is much like the command line options given when running a C compiler (e.g. -g for debug, -O4 for speed optimization). The process of selecting these options and settings is called "marking". Marks are established prior to initiating code generation ("Build Project") by editing the marking files in the /gen folder of the project and setting up the options desired.

To mark a model with options for code generation:

1. Switch to the C/C++ Eclipse perspective.
2. Edit the files ending in .mark in the /gen folder of the project.
3. Follow the documented instructions in the marking files themselves and in the BridgePoint Eclipse Help sections of the tool documentation.

Translation

BridgePoint xtUML models are translatable. This feature is the "t" in xtUML. This process is also known as "code generation." During model translation, the xtUML application model is processed and, typically, turned into code that runs on the target.

A model compiler is used to perform translation of the model from 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]. The xtUML Training materials also provide an overview of the code generation process.

Here is a high-level overview of how it works:

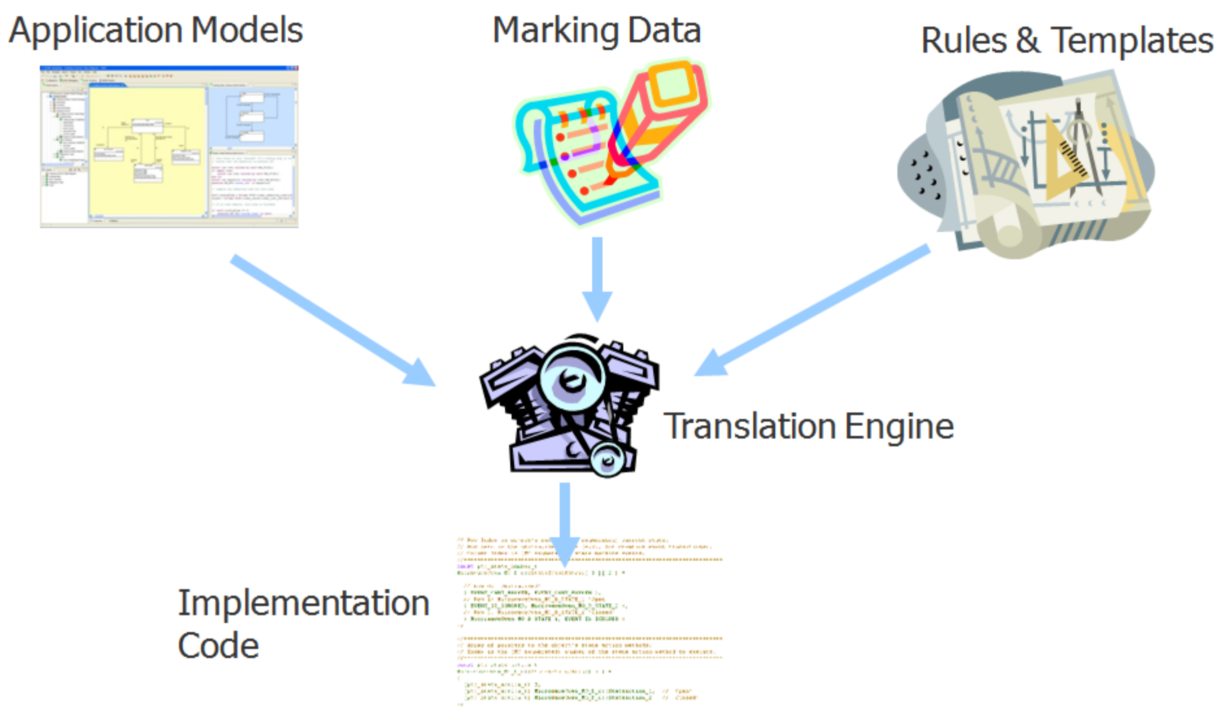


Figure 1

The process itself is not magic. There are known inputs, a tool to capture and process the inputs, and expected output. The form and function of the output is directly related to the inputs. The following sections explore each of these parts to the translation process.

Application Models

The model translation process, first and foremost, needs an input model. This is the xtUML data to be translated into a different form. There are two pieces of model data that are input into the translation engine: the project xtUML model and the metamodel.

The project xtUML model is, of course, the Executable System Model you have created inside BridgePoint expressed in xtUML models (package diagrams, class diagrams, and state machines) and Object Action Language. These define the data, control, and processing of the application.

The metamodel is a bit tricky. Here's what Mellor and Balcer have to say in [1]:

Definition: A *metamodel* is a model of a language expressed using a modeling language.

For example, the model of Executable UML is expressed using xtUML. This model of Executable UML is the so-called metamodel.

There are countless application models written in xtUML using BridgePoint. However, there is only one xtUML metamodel. The metamodel allows the translation engine to understand the style of the data to expect in the application model input, without understanding or caring about the function of the application model.

Marking

BridgePoint uses marking information to provide a means for fine tuning the translation process. This information is contained in `.mark` files located in the `gen/` folder of the project.

Marks provide implementation specific input to the translation. For example, marks are used to direct the allocation of specific classes to tasks and to map xtUML data types to data types in the implementation language. Design decisions like these do not belong in the xtUML model as they would break the platform-independent nature of the model. Neither do they belong in the implementation-agnostic translation engine.

See the Marking section of the BridgePoint Model Compiler – User's Guide in the Help system for more discussion of this topic and a detailed list of all the marks that are available.

Rules and Templates

Obviously, to translate the model from one from to another the translation engine must traverse through the application model data and convert the information it finds into the desired form. For example, an xtUML class might be converted to a struct in C and a class in SystemC. Interface messages between components may be translated into function calls in the implementation code. The rules and templates define the processing to perform on the model and the form of the output.

The rules, also called "queries", are written in BridgePoint *archetype language*. Archetype language is very similar to the OAL found inside the model. It contains commands for processing and manipulating the model data.

The templates are plain text bits of implementation language with holes that are filled with substitution variables created by the rule processing.

Here is simple archetype file that defines rule and template data. Lines that begin with "." are archetype language commands. Lines that do not are template data.

example.arc

```
.//
.// example.arc
.//
.select many obj_set from instances of O_OBJ
.for each obj_inst in obj_set
struct ${obj_inst.Name} {
  .select many attr_set related by obj_inst->O_ATTR[R102]
  .for each attr_inst in attr_set
  int ${attr_inst.Name};
  .end for
};
.end for
.//
.emit to file "example.c"
```

When this archetype file is used as input to the translation engine against a very simple model with two classes “Dog” and “Cat”, each with attributes “num_teeth” and “num_legs”, the result is:

example.c

```
struct Dog {
  int num_teeth;
  int num_legs;
};
struct Cat {
  int num_teeth;
  int num_legs;
};
```

The rules find and traverse the model data using the structure and relationships defined in the metamodel. Once the desired model data is found, the data is inserted into the templates and output as implementation code.

It is important to note that even though this process is often referred to as “model compilation” or “code generation”, the output of the translation engine (a.k.a. “model compiler”) is not restricted to implementation code. As Mellor and Balcer state in [1]:

[A]rchetypes can be written to generate anything from passive metrics on the size of the models to document macros to—if you know the syntax of the language—Klingon.

Translation Engine

The translation engine is built in to BridgePoint. In the typical scenario, it is executed when the build command is invoked inside the tool. However, the translation engine can be invoked via the command line outside of the BridgePoint environment.

As Figure 1 shows, the translation engine gathers, combines, and processes the various inputs and produces the implementation code (or other output as directed by the rules and templates).

Implementation Code

The typical output of the translation is implementation code in a language such as C, SystemC, or Ada. As discussed in the Rules & Templates section, the output of the translation could take different forms depending on your needs and the model compiler you use. See the Implementation Targets document for more information on the use and features of the BridgePoint model compilers.

No matter what implementation code or non-code output is the product of the translation process, it is important to remember that you should not modify this output. C programmers don’t modify the output of the GCC compiler. Java programmers don’t modify the .class file bytecode produced by the java compiler. Instead, they modify the source code that is input to the compilation. The same holds true for xtUML models. If an issue is found with the output from the model compilation, the appropriate input should be modified and recompiled instead of modifying the output itself.

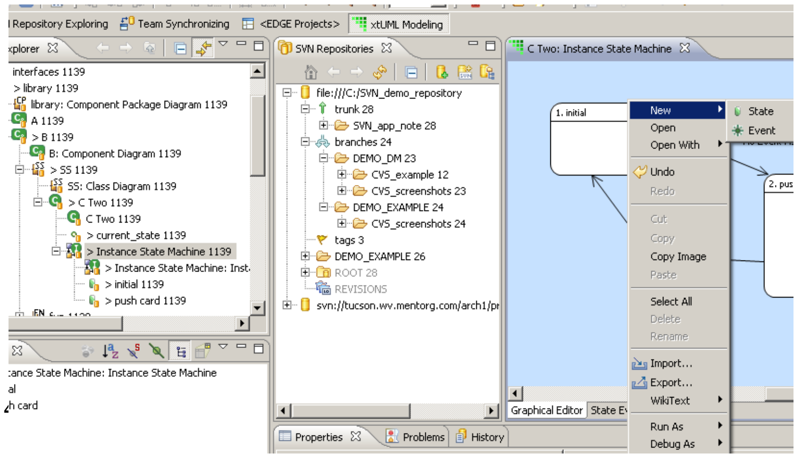
Model Management

BridgePoint is designed to operate standalone with a single modeler or in a team environment where models are concurrently developed and shared. For small projects or individual modelers, the workspace, project and model export facilities of BridgePoint are sufficient to build an effective configuration management and archival process. However, for complex projects requiring a team of modelers, the most effective BridgePoint teams supplement the native capabilities of BridgePoint with software that provides design data management, requirement tracing and configuration management. In this section, each of these capabilities will be discussed and specific recommendations in the context of the xtUML methodology will be provided

Configuration Management

Configuration management (CM) is the set of practices and software that allow multiple team members to simultaneously work without interference on a common set of model elements. At its core, CM involves a data repository which registers with each update a summary comment intended to document incremental changes and traceable links to requirements that motivated the change. To enable concurrent development, the CM technology is complemented with a set of practices that define how changes made in a solution branch are merged back into the main implementation. There are a range of software solutions that provide these additional capabilities, and BridgePoint is designed to work with all solutions that support the standard Eclipse framework (TEAM interface). For our purposes, this discussion will be based on the Subversion repository and its nomenclature.

Overview of the Recommended Process



Central to the Recommended Process is the idea of Requirements-driven Model development. Under this approach, a project begins based on an identified set of requirements that must be satisfied at completion. Each requirement should represent a cohesive effort, whether it’s a new capability or the repair of a defect. Often, some of these requirements will involve resolving defects or issues present in earlier versions of the project. The collection of requirements are then selected one at a time on a priority basis, and assigned to a modeler. It is at this stage, that the CM process becomes involved.

Once a requirement is assigned, the modeler creates a copy of the golden source, and places it in a work-in-progress area of the CM repository. It is common to call the golden source, the TRUNK version, and the work-in-progress copy, the BRANCH version. Modelers then introduce their changes to the BRANCH copy until the requirement is fully implemented and tested. In a team setting where multiple requirements are simultaneously being worked on, it is possible for the TRUNK version to change while other requirements are still under development. When this occurs, the working BRANCH for each of these in-progress requirements must be SYNCHRONIZED with the revised TRUNK version. This ensures that when a requirement is completed, the resulting BRANCH can be easily MERGED with the current TRUNK version.

The individual steps for each of these tasks will be omitted in this Overview, since they are well documented in the Application Note “BridgePoint and Configuration Management Using Subversion” and the BridgePoint User Manual.

Requirements Tracking

This section describes a recommended approach for tracking project progress through the BridgePoint development process. The BridgePoint process follows four steps: Analysis, Design, Verification, and Code Generation. These steps are described in detail elsewhere. This section describes a method for capturing and tracing requirements as they are worked-on and satisfied during the BridgePoint process.

Prerequisites

1. Understand the recommended BridgePoint documentation practices. This topic is described by Documentation-GeneratingAndUsingDocumentation.docx.
2. Understand how to use a revision control system with BridgePoint models. This topic is covered in the BridgePoint Reference in eclipse under "Help > BridgePoint UML Suite Help > Reference > BridgePoint and Configuration Management".
3. Understand how to use an issue tracking system or a requirements management system to monitor required tasks.

    - Examples of issue tracking systems are ClearQuest and Bugzilla. However, there are many such systems and the choice is driven by business needs of your company.

Capture Requirements

Requirements are gathered from many sources. Each defined requirement is associated with a unique defined requirement identifier. In some workplaces, there is a single system used for defining requirements and tracking the requirements. In this example we will assume there is a separate system for requirements creation and tracking. In our example, the requirements are captured in a simple revision controlled document. Here is an example of what one section of this document may look like:

Functional Requirements Group 1

Section/ Requirement ID	Requirement Definition
FR1.0.	The system shall [parent requirement group 1].
FR1.1	The system shall [child/parent requirement].
FR1.1.1	The system shall [child requirement].
FR1.1.2	The system shall [child requirement].

Associating Tasks with Requirements

For each requirement called out in the requirements document, an issue is raised in the requirements tracing system.   There may be many tasks for a single requirement and there may be many requirements dependent on a single task. For each task raised there must be a unique task identifier. In this example, the unique issue identifier provided by requirements tracing system is used as this task identifier. When we enter the task a required field is defined in requirements repository that holds the requirement id.

Trace Requirements

After creating the tasks as described above, we have a clear mapping of the tasks to the associated requirements. This allows the issue tracking system, to be used as means to report progress on the project. After entering some tasks, here is an example of what a report from the issue tracking system showing the status of each issue can look like:

Requirement_ID	Task_ID	State	Headline
FR1.0	dts01001234	assigned	Task associated with requirement FR1.0
FR1.0	dts01001235	new	Another Task associated with requirement FR1.0
FR1.0	dts01001236	resolved	Yet another Task associated with requirement FR1.0
FR1.1	dts01001237	assigned	Task associated with requirement FR1.0
FR1.1	dts01001238	assigned	Another Task associated with requirement FR1.0
FR1.1	dts01001239	assigned	Yet another Task associated with requirement FR1.0

Use Commit-time Scripting

Virtually every configuration management tool provides "commit-time scripting". This is simply a hook into the tool to run scripts as a gate to committing changes into the database. Commonly, this hook is used to run lint or other source code checkers. Some organizations compile the source and run unit tests as a gate to commitment into the configuration management system.

In the example used in this document, there is a bridge that runs between the configuration management system and the issue tracking system. This bridge forces developers to associate the source code being committed to the issue that precipitated the change.

- This bridge updates the issue tracking system for each check-in made. This update includes the file(s) changed and the description of the change.

This mechanism assures that no modification to the revision control system occurs without a proper task and requirement associated with the change.

This script is also used to email an appropriate group to call attention to the change.

- This can lead to a lot of emails from the system; however they are very easily filtered and sent to appropriate folders by your email application.
- This allows the group to monitor the process at a very isolated level if desired.
- It is suggested that team members be trained to enter quality comments when they check-in changes. These comments are captured in the issue tracking system where they can be used, as needed, to document the all the changes made for any particular task.

Trace Requirements into the Model Elements

Using commit-time scripting as described above provides a link from each change made for any task to the ".xtuml" file the change was made against. This link is present because when the change is checked-in to revision control, commit-time scripting updates the task in the issue tracking system to capture this data. Commit-time scripting alone does NOT track the change made to a particular model element inside the ".xtuml" file.

- To track changes into the model element(s) themselves the model element's description attribute is used.
- Each time a change is made to a model element, the engineer adds the task id to the description field before checking the change into the revision control system.
- The task id is now stored in the model itself. It is in the description attribute of the element changed.

Trace Requirements into the Generated Artifacts

The section above, Trace requirements into the model artifacts, describes how to use the model element's description attribute to trace requirements in to the model elements themselves. Once that is done, a model compiler can extract this data from the description attribute into the generated artifacts.   This provides a huge amount of flexibility to use this data in external reports or other generated artifacts.

The generated artifacts that can include these task identifiers can include virtually anything. They can include source code, reports, interface descriptions, and more.

Monitoring Progress with Reports

As described above in the Tracing Requirements section, most progress reporting can be done by utilizing the issue tracking system's built-in reporting. Additionally, custom reports on particular aspects of the system can be created using a "Report Generator Model Compiler". This type of report generation is described in the documentation section titled: "Creating a Report Generator Model Compiler".

Documentation

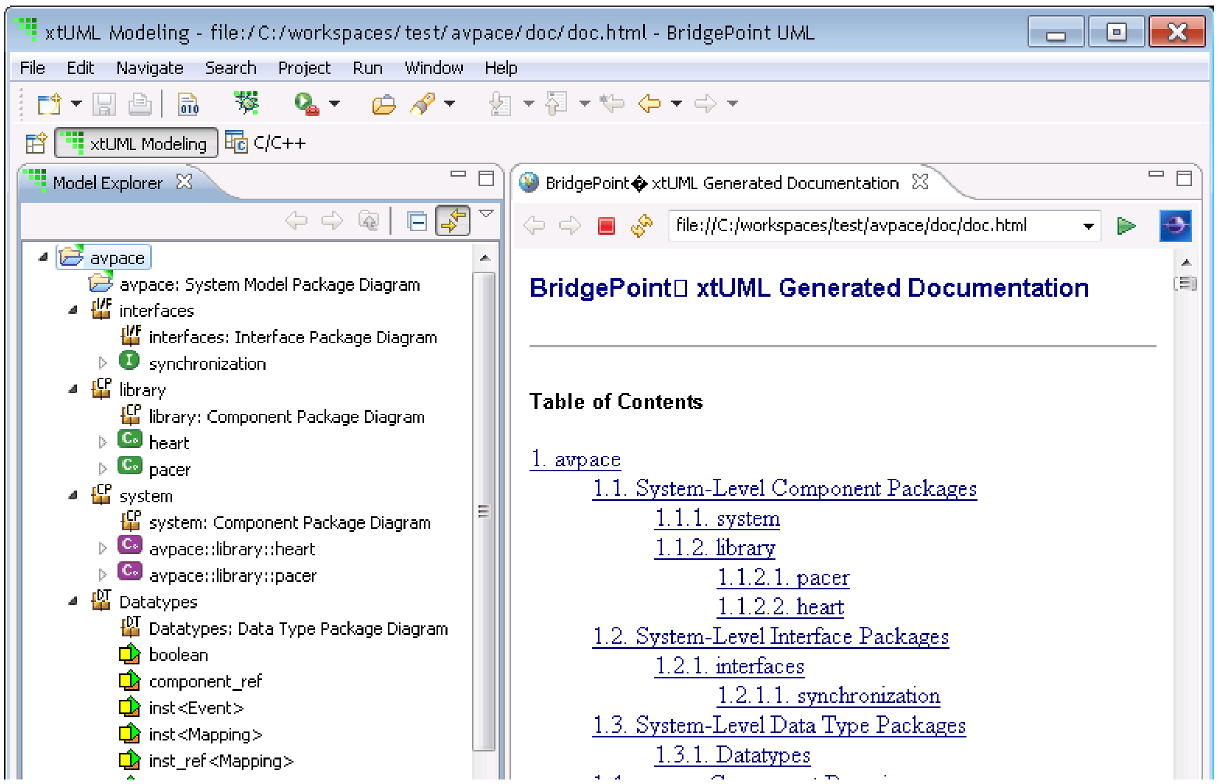
This section describes how to capture documentation from BridgePoint models and how to use that documentation in external documents.

When to Capture Documentation

Documentation can be captured at any stage of the project cycle. Examples include:

- Capture the state of the entire model and report on project progress.
- Share the status of a particular piece of the model (see Copy/Paste directly from your model). This is useful to elicit feedback before checking changes into revision control, or to get feedback without requiring colleagues to extract model changes from a source repository.

Generating the Documentation (DocGen)


xtUML Modeling - file:/C:/workspaces/test/avpace/doc/doc.html - BridgePoint UML
File
Edit
Navigate
Search
Project
Run
Window
Help
xtUML Modeling
C/C++
Model Explorer
BridgePoint� xtUML Generated Documentation
file://C:/workspaces/test/avpace/doc/doc.html

avpace
- avpace: System Model Package Diagram
- interfaces
  - interfaces: Interface Package Diagram
  - synchronization
- library
  - library: Component Package Diagram
  - heart
  - pacer
- system
  - system: Component Package Diagram
  - avpace::library::heart
  - avpace::library::pacer
- Datatypes
  - Datatypes: Data Type Package Diagram
  - boolean
  - component_ref
  - inst<Event>
  - inst<Mapping>
  - inst_ref<Mapping>

BridgePoint□ xtUML Generated Documentation

Table of Contents

Figure 2

BridgePoint contains a feature called DocGen. DocGen generates documentation for your model. Documentation can be generated at any stage of the project development cycle to capture the state of the model and report on project progress:

Launching DocGen

1. Open the xtUML Modeling Perspective
2. Select the project you want to generate documentation for
3. Right Click and select "Generate documentation" from the context menu (see figure 2).
4. The documentation is generated into the "<project>/doc" folder under the selected project.

Figure 3

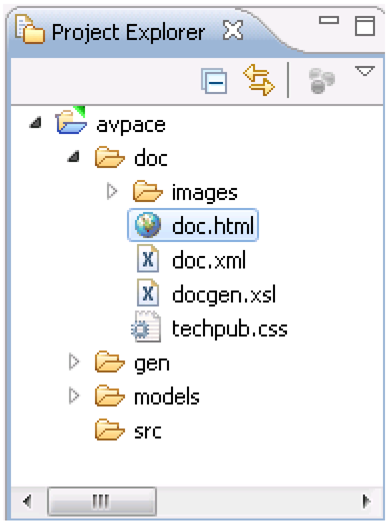
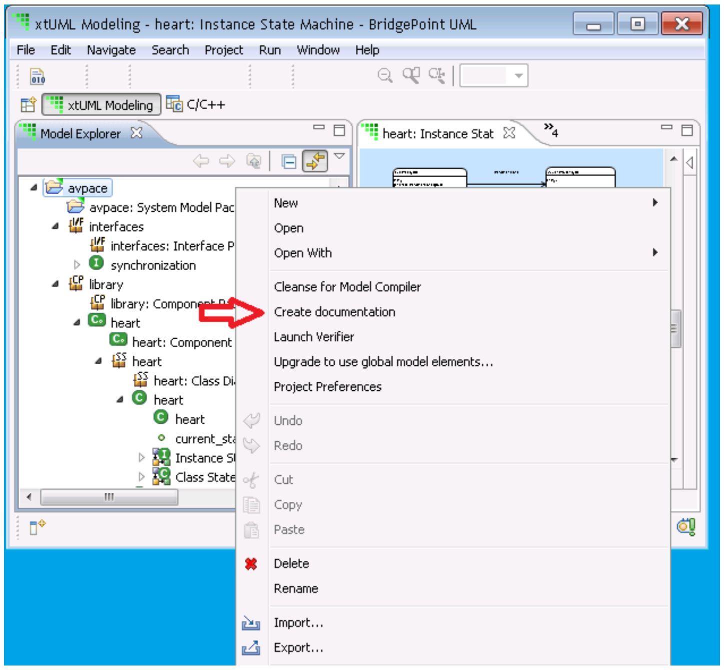

Project Explorer
avpace
doc
images
doc.html
doc.xml
docgen.xsl
techpub.css
gen
models
src

Figure 4


xtUML Modeling - heart: Instance State Machine - BridgePoint UML
File Edit Navigate Search Project Run Window Help
xtUML Modeling
C/C++
Model Explorer
heart: Instance Stat
avpace
avpace: System Model Pac
interfaces
interfaces: Interface P
synchronization
library
library: Component P
heart
heart: Component
heart
heart: Class Di
heart
heart
current_sta
Instance S
Class State
New
Open
Open With
Cleanse for Model Compiler
Create documentation
Launch Verifier
Upgrade to use global model elements…
Project Preferences
Undo
Redo
Cut
Copy
Paste
Delete
Rename
Import…
Export…

Navigating the Generated Documentation

After generating documentation with DocGen the result is placed in a folder named “doc” under the eclipse project folder. The generated documentation is in html format. The content of this documentation is generated into a file named “doc.html” (see figure 3). Images are captured of the model elements, which are stored in GIF format in <project>/doc/images

When doc.html is opened the first item displayed is a Table of Contents (see figure 1). Every item in the Table of Contents is an html link to a section of the document. The generated documentation is organized as follows:

- Table of contents
    - System-level Component Packages
    - System-level Data type packages
    - OAL Activity listings
        - State Actions
        - Class Operations
        - Functions
        - Bridge Operations
        - Required Operations
        - Provided Operations
        - Required Signals
        - Provided Signals
    - List of Figures
        - This section contains a link to every diagram in the model.
    - List of Tables
        - This section includes links to tables generated from the model data. The data in these tables is a tabular format for the data represented by diagrams in the model.

Following the Table of Content the data referred to by the links in the table of contents is list, one section at a time, in the same order it is presented in the table of contents.

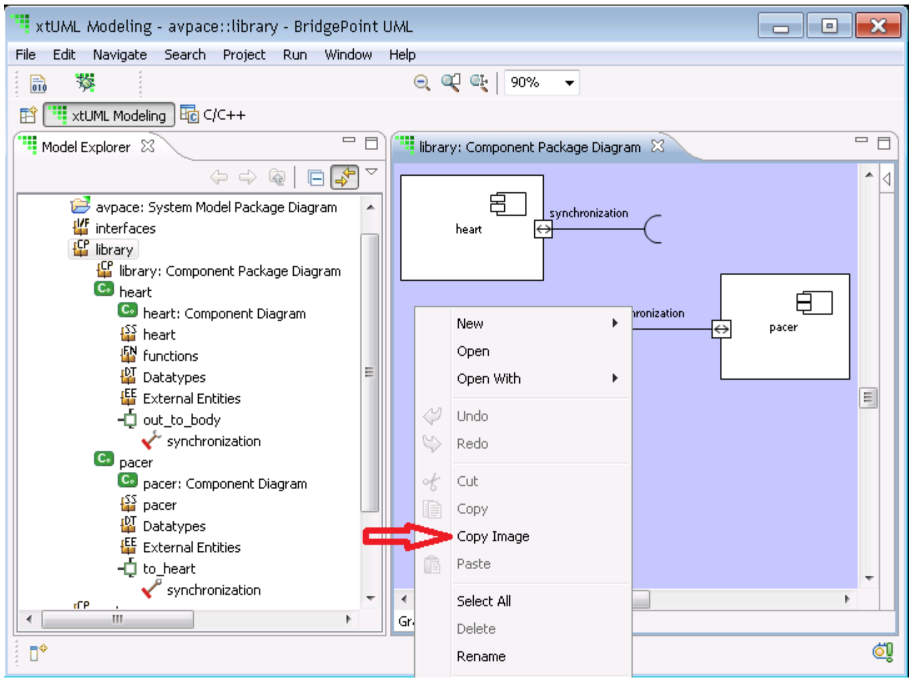
Using the Generated Documentation in External Documents

There are several options for copying documentation from you model to an external document. It is important to note here that you do NOT have to use DocGen to capture and include your diagrams in external document. BridgePoint gives you the ability to copy and paste these directly from the canvas:

Copy/Paste directly from your model to the destination

1. Open Model Explorer
2. Open the diagram being copied on the canvas by double-clicking it in Model Explorer
3. Select the desired images
4. Select one or more model elements on the canvas by selecting the desired items using <ctrl>-<left mouse click>
5. If desired, Use <ctrl>-A or Edit ->Select All to select all the elements
   HINT: Simply right clicking on an empty spot on the canvas, will select all the images.
6. Right Click the selection and choose "Copy Image"

Figure 5



Note: Do NOT select “Copy”. “Copy” copies the model instances so they may be pasted into another model. “Copy Image” copies only the image data. Therefore, “Copy Image” is the desired option in this situation.

Paste the result into any standard work processing application. For the example shown above in Figure 4, Figure 5 below shows the pasted result:

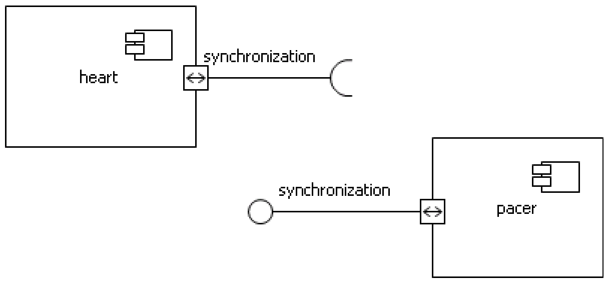


Figure 6

Copy/Paste from the generated Document to the destination

See Launching DocGen for a description of how to generate documentation for your model

1. Open doc.html
2. From the table of contents navigate to the desired section
3. Copy the desired select
4. Paste the selected data into your destination document

Provide a HTTP Link to Generated Documentation

See Launching DocGen for a description of how to generate documentation for your model.

See Navigating the Generated Documentation for a description of the layout of the generated documentation.

Copy the doc folder and its contents to an internal web server that you can provide link to. The following are examples of shared locations:

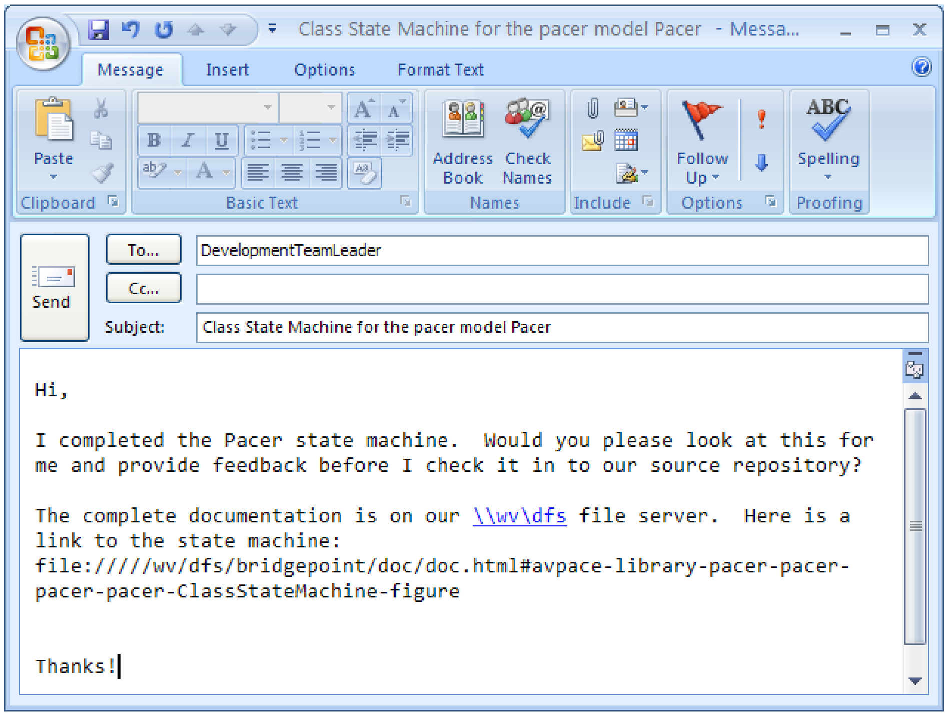
- File (Documents are on a file server):
- file://///wv/dfs/bridgepoint/doc/doc.html
- HTTP URL (Generated documents are placed on a web server):
- http://tucson.wv.mentorg.com/BridgePoint/R340/doc.html

This is an example that links to a particular artifact in the generated documentation:

- http://tucson.wv.mentorg.com/BridgePoint/R340/doc.html#avpace-library-pacer-pacer-pacer-pacer-ClassStateMachine-figure

Embed links to this document in other documents, emails, etc…

Figure 6 in an example where a team member who is working on a project is using the generated documentation to get feedback on the work he performed. Figure 7 is what the reader of the email will see when they follow the link.



To...: DevelopmentTeamLeader

Subject: Class State Machine for the pacer model Pacer

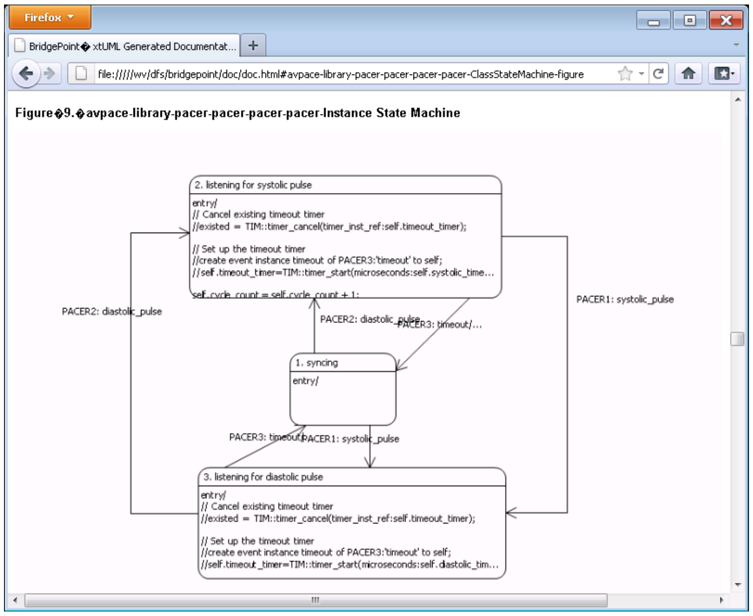
```
Hi,

I completed the Pacer state machine.  Would you please look at this for
me and provide feedback before I check it in to our source repository?

The complete documentation is on our \\wv\dfs file server.  Here is a
link to the state machine:
file://///wv/dfs/bridgepoint/doc/doc.html#avpace-library-pacer-pacer-
pacer-pacer-ClassStateMachine-figure


Thanks!
```

Figure 7



file://///wv/dfs/bridgepoint/doc/doc.html#avpace-library-pacer-pacer-pacer-pacer-ClassStateMachine-figure

Figure�9.�avpace-library-pacer-pacer-pacer-pacer-Instance State Machine

```
2. listening for systolic pulse
entry/
// Cancel existing timeout timer
//existed = TIM::timer_cancel(timer_inst_ref:self.timeout_timer);

// Set up the timeout timer
//create event instance timeout of PACER3:'timeout' to self;
//self.timeout_timer=TIM::timer_start(microseconds:self.systolic_time...

self.cycle_count = self.cycle_count + 1;
```

```
1. syncing
entry/
```

```
3. listening for diastolic pulse
entry/
// Cancel existing timeout timer
//existed = TIM::timer_cancel(timer_inst_ref:self.timeout_timer);

// Set up the timeout timer
//create event instance timeout of PACER3:'timeout' to self;
//self.timeout_timer=TIM::timer_start(microseconds:self.diastolic_tim...
```

- Figure 8

Creating a Report Generator Model Compiler

This section describes how to create a simple archetype that defines a report generated based on the data in a model. By adding to the archetype, this simple example can be expanded to create any report.

Prerequisites

- An archetype is a fragment of data access and text manipulation logic. It is written in RSL. Please see the RSL guide in the BridegPoint documentation to learn how to write RSL. This section assumes the reader is familiar with basic RSL concepts.
- To take advantage of BridgePoint ability to execute an arbitrary archetype that is placed in the projects "gen" folder, a source model compiler license must be used. See the BridgePoint model compiler guide for information on using a source model compiler.

Creating a simple report

When the Project -> Build Project option is selected, BridgePoint will look for an archetype (*.arc) file in the gen folder of your project. If it finds one, it will load the model into the "gen database", run the archetype, and then exit without finishing the rest of the model compilation process.

- Create an archetype that defines your desired report.

  The following is very simple archetype that simply outputs the

```
.//
.// classes.arc
.// Simple example that finds all the classes in the model
.// and writes their names to a file named "classes.txt"
.//
Here are all the classes in this model:
.select many obj_set from instances of O_OBJ
.for each obj_inst in obj_set
  Object name is ${obj_inst.Name}
.end for
End of List of Object Names
.//
.emit to file "classes.txt"
```

- Open the C/C+ perspective
- Rick-click the project and select “Build Project”
- The resulting file, “classes.txt” in this example is placed in default folder, “gen”, since no explicit path was specified with the file name.

Recommendations

- DocGen does not currently provide options to allow different style sheets or custom output of any type. However, by creating a Report Generator it is possible to create the report in any desired form.
- DocGen output can be mixed into a custom report. For example, the custom report archetype could refer to the images captured by docgen.

References

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