

*An Illustrated Example using Enterprise Architect*

# *Embedded Systems Development using **SysML***

*By Doug Rosenberg  
with Sam Mancarella*



## Table of Contents

|           |   |           |
|-----------|---|-----------|
| Prologue  | – <i>Back to the Future</i>   | <b>3</b>  |
| Chapter 1 | – An Introduction to SysML and Enterprise Architect Engineering Edition | <b>4</b>  |
| Chapter 2 | – Audio Player Requirements   | <b>14</b> |
| Chapter 3 | – Audio Player Behavior   | <b>18</b> |
| Chapter 4 | – Audio Player Structure  | <b>27</b> |
| Chapter 5 | – Audio Player Constraints and Parametrics                              | <b>32</b> |
| Chapter 6 | – Audio Player Hardware Implementation                                  | <b>39</b> |
| Chapter 7 | – Audio Player Software Implementation                                  | <b>54</b> |

# Prologue - Back to the Future

This book represents a departure from what I've been doing at ICONIX over the last 25 years, which has been focused mainly on Software Engineering, but a book on how to develop real-time embedded systems is actually a "return to my roots" in Electrical Engineering.

My degree is actually in EE, not Computer Science, and my path to being involved with Software Engineering had its formative years when I was working as a programmer in Computer Aided Design for VLSI in the aerospace industry in southern California, and also up in Silicon Valley. So even though SysML is a new area for me, I'm inherently familiar with the problems that SysML helps to solve.

My first four jobs out of college (in the early 1980s) were involved with VLSI design. Three of these (two at TRW and one at Hughes Research Labs) were on a project called VHSIC (Very High Speed Integrated Circuits), which is what the "V" in VHDL stands for. At TRW my work involved extending some of the early "design rule checking" software to cover a more complex fabrication process that allowed us to deliver gigahertz-level speeds, which was much more of an accomplishment 30 years ago than it is today. I also worked a bit with SPICE, one of the earliest "circuit simulators" (more about simulation in the "SysML parametrics" discussion in Chapter 5).

Later, after a short stint up in Silicon Valley working on something called "symbolic layout and compaction" at a company called Calma, I returned to TRW where I designed and programmed an application called "Hierarchical Layout Verification" which recursively decomposed a complete integrated circuit layout into sub-cells (now called "blocks" in SysML), determined their input and output "ports" (another familiar SysML concept), and checked both physical design rules and electrical connectivity.

During this time, my boss Jim Peterson at TRW was developing one of the early Hardware Description Languages, which he called THDL (for TRW Hardware Description Language). THDL itself was an extension of CIF (Caltech Intermediate Format<sup>1</sup>) which had been developed in Carver Mead's research group when Jim was a grad student at Caltech. Since Jim's THDL work was funded under the VHSIC contract it's a safe bet that some of the concepts in VHDL had their roots in THDL.

After my second go-round at TRW, I went to work at Hughes Research Labs in Malibu, CA, developing the interface from CAD systems (most notably Calma, who owned about 80% of the market back then) to something called the VHSIC Electron Beam Lithography System. This was another ambitious project that pushed the state of the art in fabrication technology far ahead of what it had been previously. We were writing one-tenth-of-a-micron lines on silicon wafers using electron beams (still not bad today) back in 1984.

When Sparx Systems asked me to write this eBook, I discovered a kindred spirit in Sam Mancarella, who is largely responsible for a great deal of the implementation of Enterprise Architect's SysML solution. Sam also developed the Audio Player example that this book is written around, which is such a complete and comprehensive example that it made my writing task very easy. I want to make it completely clear that **Sam deserves ALL of the credit for developing this example**, and that my contribution to this project was simply writing the manuscript around the example. My electrical engineering background made it obvious to me how good Sam's example is, and allowed me to see how the pieces fit together.

---

<sup>1</sup> Introduction to VHDL By R.D.M. Hunter, T.T. Johnson, p.17-18

# Chapter 1 - An Introduction to SysML and Enterprise Architect Engineering Edition

## ***A roadmap for embedded system development***

It's easy for a book to present a taxonomy of disjointed SysML diagrams and then leave you to figure out how to combine those diagrams into a meaningful model. In fact, that's what the majority of SysML books that we've seen appear to do. But with this book, we're going to introduce you to SysML and the Systems Engineering Edition of Enterprise Architect in a rather different way.

At ICONIX, we've had pretty good success when we defined an unambiguous development process, and presented that development process in "roadmap" form. We've developed *process roadmaps* for use case driven software development, business modeling, design-driven testing, and algorithm-intensive software design. In this book we're going to do it again, this time for embedded systems that involve a combination of hardware and software. We'll explain the roadmap at the high level in this chapter, and then each of the following chapters will detail one of the high-level activities on the top-level roadmap. Along the way, we'll show you how Enterprise Architect's System Engineering Edition supports the process we're describing, while illustrating each step of the process by example.

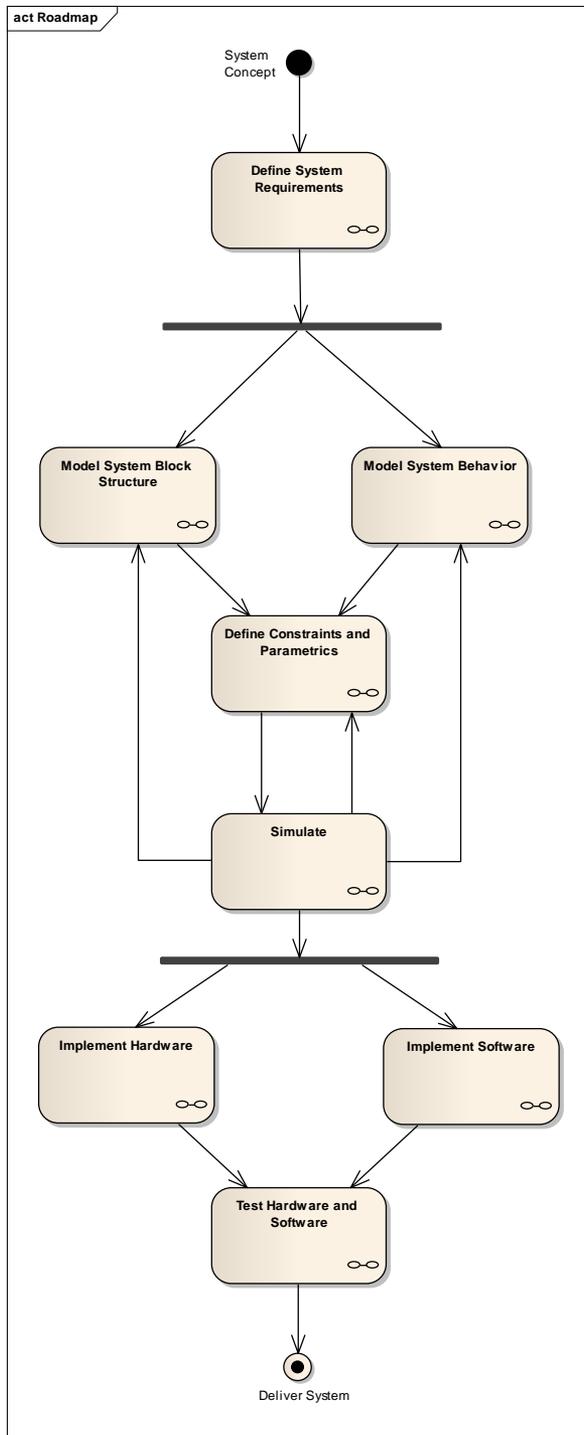
In addition to providing complete support for all SysML 1.1 diagrams, the Enterprise Architect Systems Engineering edition combines advanced features such as executable code generation from UML models (including support for hardware languages such as Verilog and VHDL), executable SysML Parametric diagrams and advanced scripting. We'll explore this unique combination of advanced capabilities in the last half of this book.

Specifically,

- In Chapter 5 we'll explore Enterprise Architect's SysML Simulation Support, which provides the capability of simulating SysML 1.1 constraint models with results graphing capabilities;
- In Chapter 6 we'll describe support for Hardware Description Languages, including Verilog, VHDL and SystemC, with support for generating State Machine code; and
- In Chapter 7 we'll illustrate Enterprise Architect's support for generating functional source code for State Machines, Interactions and Activities in C, C++, C#, Java and VBNet .

Each of these capabilities, taken standalone, adds a significant amount of "horsepower" for a systems engineering effort. We'll show you how to combine these capabilities into a single process roadmap that's greater than the sum of its parts.

Figure 1 shows the top level roadmap for ***ICONIX Process for Embedded Systems***.

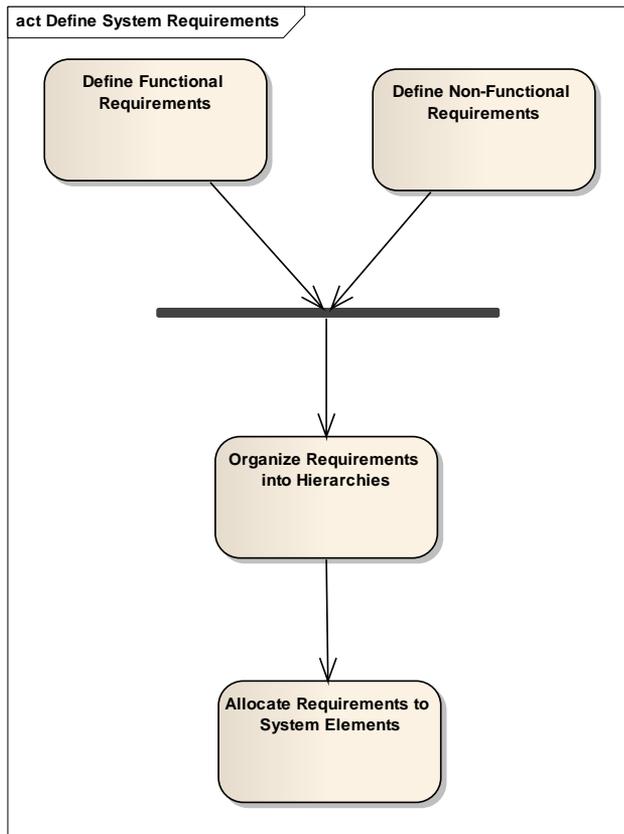


**Figure 1 – ICNIX Process Roadmap for Embedded Systems Development**

As you can see, our roadmap starts off by defining requirements, proceeds through modeling of system behavior and block structure, and then through definition of constraints and parametrics, simulation, and then implementation in both hardware and software. We'll take you through each of these activities at a summary level in this chapter, and then in more detail, illustrated by a comprehensive Audio Player example, in Chapters 2-7.

## **Requirements, Structure, Behavior, and Parametrics – the Four Pillars of SysML**

Our Embedded Systems Development Process Roadmap is organized around producing a SysML model that is generally organized into four sections. These parts of the overall system model (Requirements, Structure, Behavior, and Parametrics) are sometimes referred to as “The Four Pillars of SysML”.<sup>2</sup>



**Figure 2 – Roadmap: Define System Requirements**

### **Requirements**

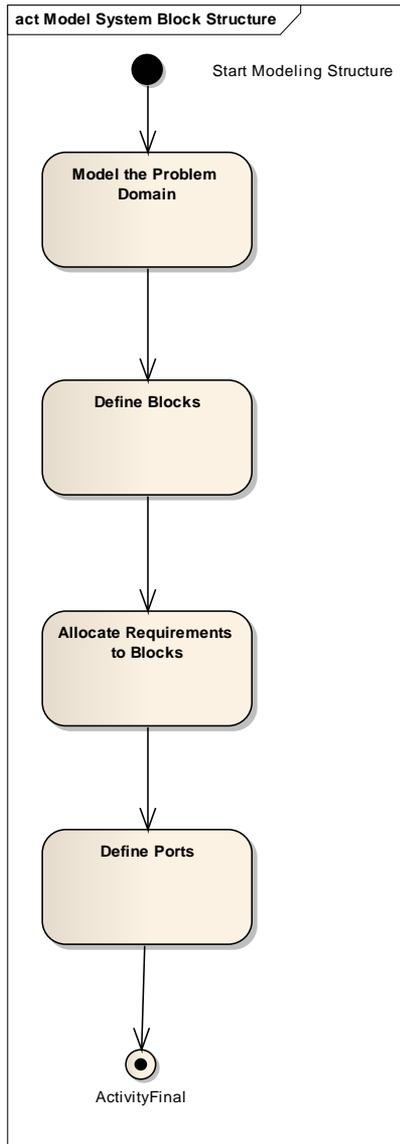
Requirements are generally categorized as Functional Requirements, which represent capabilities of a system, and Non-Functional Requirements, which cover such areas as Performance and Reliability. You can organize Requirements into hierarchies on requirement diagrams. Enterprise Architect supports allocation of requirements to other elements using a simple drag-and-drop, and automatic generation of traceability matrices.

Figure 2 shows the steps for Requirements definition from our process roadmap. Note that allocation of Requirements to System Elements is really an ongoing process as the model is developed, and largely occurs within other roadmap activities. We’ll explore Requirements Definition in more detail in Chapter 2.

---

<sup>2</sup> OMG Systems Modeling Language Tutorial, INCOSE 2008

## Structure



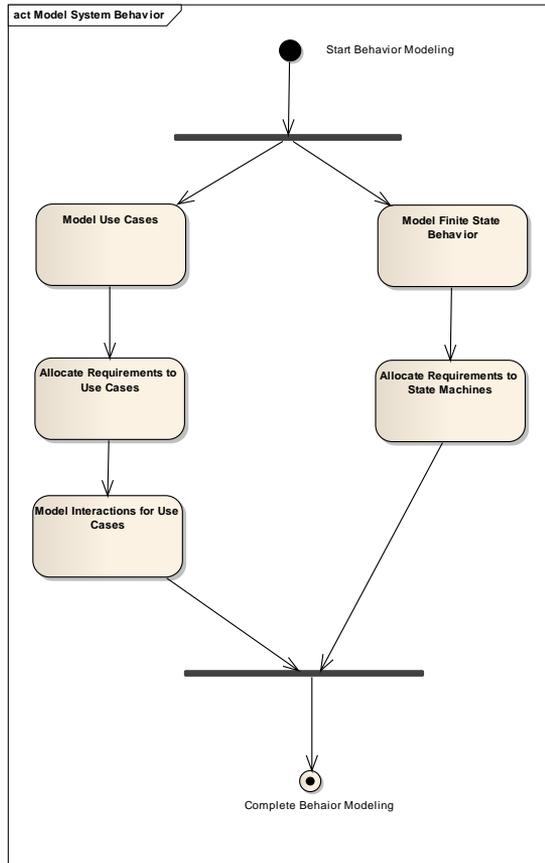
**Figure 3 – Roadmap: Model System Block Structure**

Blocks can be used to represent hardware, software, or just about anything else. Block definition diagrams represent system structure. Internal block diagrams describe the internals of a block such as parts, ports, and connectors. As with UML, Packages are used to organize the model.

If you think of a Block as an electronic circuit (one of many things that a Block can describe), the Ports define the input and output signals to/from the circuit. SysML allows you to describe the input signals and transformations in great detail, and Enterprise Architect contains a built-in simulator that allows you to plot the output graphically or export to a comma-separated value file. You'll see how this works in detail in Chapter 5. Defining the Block structure is a prerequisite for defining parametrics and running simulations.

Here's how our process roadmap approaches defining system structure. See Chapter 3 for an expanded discussion on modeling structure.

# Behavior



**Figure 4 – Roadmap: Model System Behavior**

SysML provides four main constructs to represent different aspects of system behavior; use cases, activity diagrams, sequence diagrams, and state machines.

Our roadmap shows two parallel branches for modeling system behavior. One branch starts with use cases<sup>3</sup>, which describe scenarios of how users will interact with the system. Use cases generally consist of a “sunny-day” part which describes a typical success-path for the scenario, and multiple “rainy-day” parts which describe unusual conditions, exceptions, failures, etc. Use cases are typically detailed on Interaction (Sequence) Diagrams.

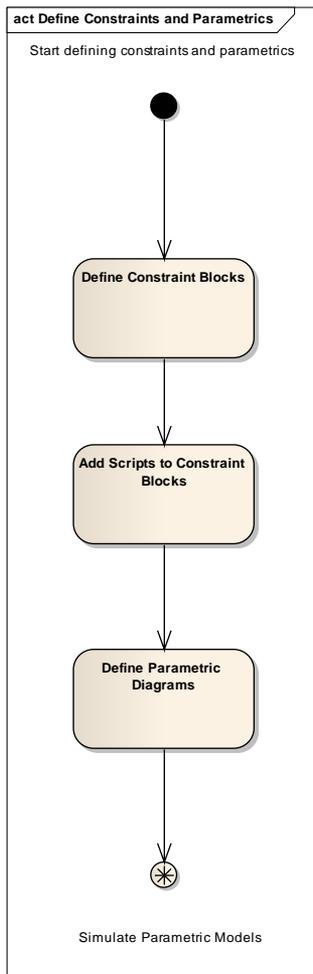
The other branch on the roadmap involves defining event-driven, finite-state behavior of some part of a system using state machines. As a simple example, there is finite state behavior associated with the power charging circuitry on our Audio Player. One of Enterprise Architect’s unique capabilities is the ability to generate functional (algorithmic) code from state machines. As you’ll see, these state machines can be realized in software or in hardware using Hardware Description Languages (HDLs).

Requirements are allocated to both use cases and states. Chapter 4 explores behavior modeling in detail.

---

<sup>3</sup> See **“Use Case Driven Object Modeling with UML: Theory and Practice”** by Doug Rosenberg and Matt Stephens for a lot more information about use cases.

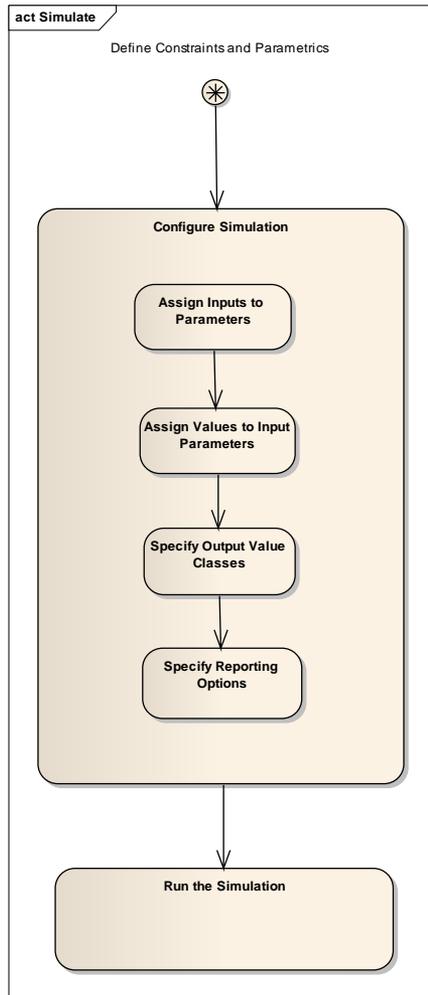
## Advanced Features of the Enterprise Architect System Engineering Edition



**Figure 5 – Roadmap: Define Constraints and Parametrics**

Enterprise Architect Systems Engineering edition contains a number of unique features for systems and software engineers working on embedded systems. The Systems Engineering edition combines new features such as executable SysML Parametric diagrams and advanced scripting with executable code generation from UML models (including support for hardware languages such as Verilog and VHDL, and bundles licenses for DoDAF-MODAF, SysML, DDS and IDE integration products to provide powerful model-driven construction tools to tightly bind your code development in Eclipse or Visual Studio with the UML/SysML. Our process roadmap leverages these unique capabilities into a synergistic development process.

## Parametrics



**Figure 6 – Roadmap: Simulate**

Parametrics allow us to define detailed characteristics, physical laws, and constraints on system blocks that allow us to simulate how a system will behave, then make engineering tradeoffs, and re-simulate until our design meets the specified requirements.

Our roadmap provides two high-level activities in this area; the first to define constraint blocks and parametric diagrams, and the second to configure and execute the simulations.

The ability to configure and execute simulations within Enterprise Architect, eliminating the need to export the model to external simulation software, is one of the unique capabilities of the Sparx Systems SysML solution.

Enterprise Architect's built-in support for scripting and graphical display of simulation results tightens the feedback loop on making engineering tradeoffs in the model to rapidly ensure that all system requirements are met. You'll see how this works in detail in Chapter 5.

## ***Implement Hardware***

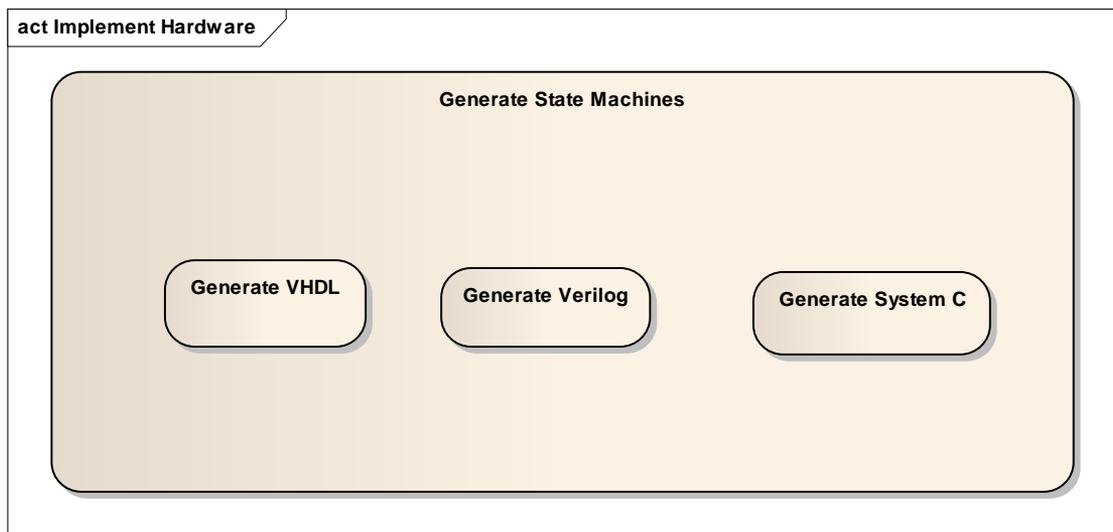
Hardware Description Languages allow the specification of electronic circuits in a software-

like representation. According to Wikipedia<sup>4</sup>:

*In electronics, a **hardware description language** or **HDL** is any language from a class of computer languages and/or programming languages for formal description of electronic circuits, and more specifically, digital logic. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.*

*HDLs are standard text-based expressions of the spatial and temporal structure and behaviour of electronic systems. Like concurrent programming languages, HDL syntax and semantics includes explicit notations for expressing concurrency. However, in contrast to most software programming languages, HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as netlist languages used on electric computer-aided design (CAD).*

*HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being programming languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.*



**Figure 7 – Roadmap: Implement Hardware**

Enterprise Architect's long-proven ability to generate code has been extended to support code generation in VHDL, Verilog, and SystemC in the Systems Engineering Edition. While code generation is independent of SysML usage, from a process roadmap standpoint, this means we can drive both hardware and software implementation from our SysML model. Once code is generated in an HDL, it's possible to "compile to silicon" to realize the hardware solution on a chip.

We'll explore hardware implementation in Chapter 6.

---

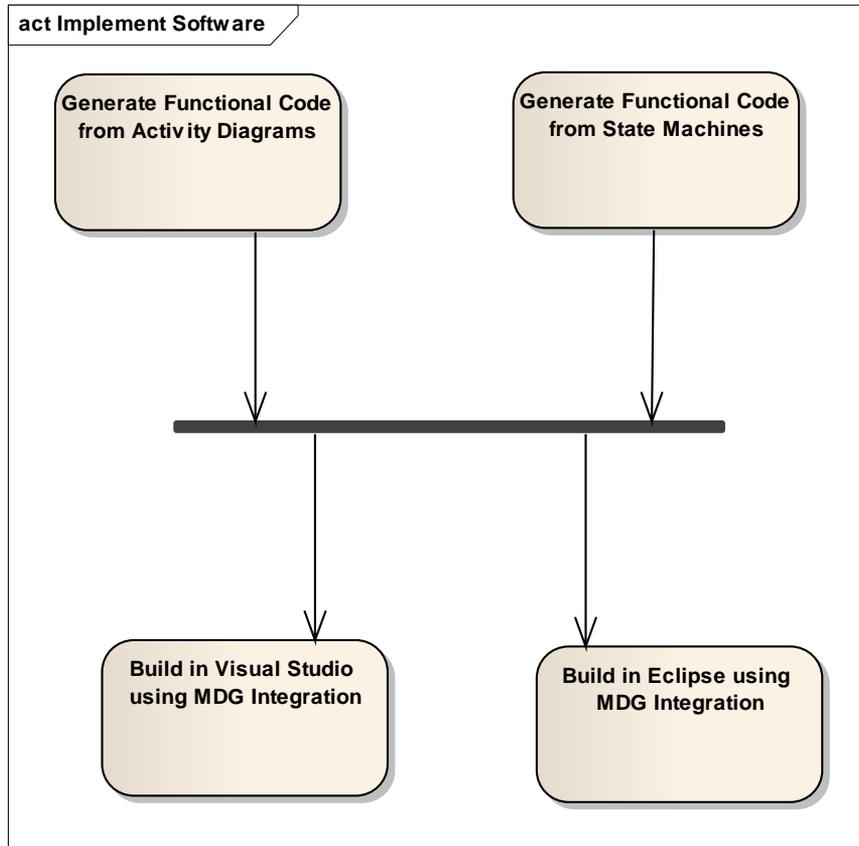
<sup>4</sup> [http://en.wikipedia.org/wiki/Hardware\\_description\\_languages](http://en.wikipedia.org/wiki/Hardware_description_languages)

## ***Implement Software***

Software implementations can leverage a variety of powerful capabilities that are included with the System Engineering Edition of Enterprise Architect. Two of the more important and unique capabilities are:

- The ability to generate functional (algorithmic) code from behavioral models (state machines, activity diagrams, and interaction diagrams)
- The ability to integrate Enterprise Architect models into development environments such as Eclipse and Visual Studio.

Figure 8 shows a high-level look at the Software Implementation activity from the roadmap.



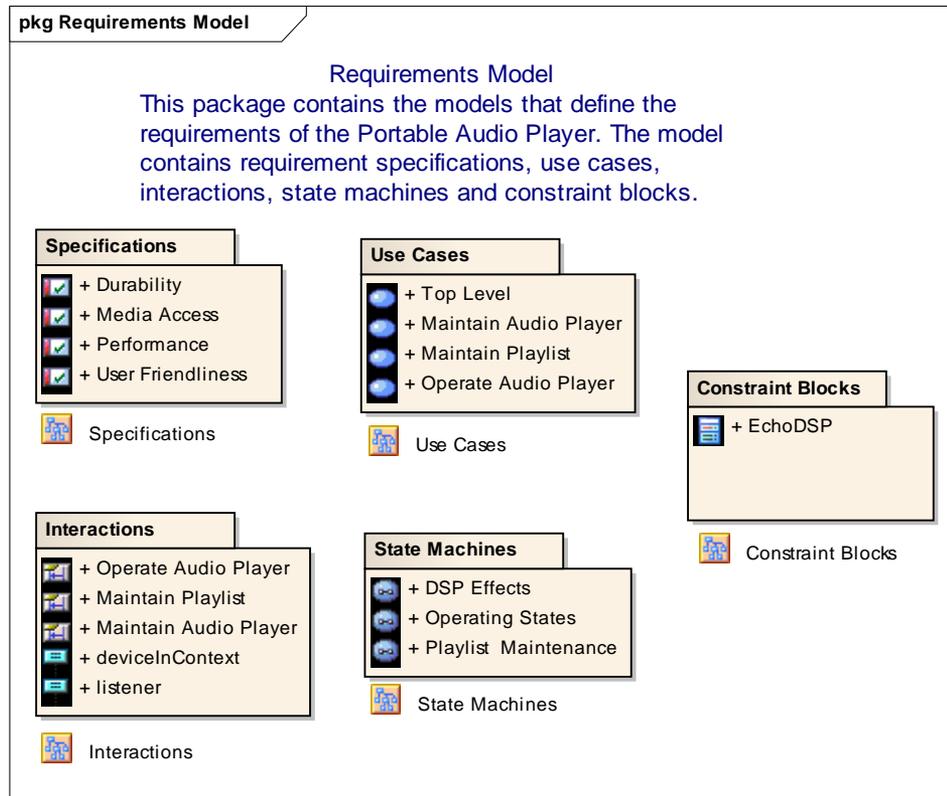
***Figure 8 – Implement Software***

We'll explore these unique capabilities and how they work together in Chapter 7.

## Introducing the Audio Player Example

Over the course of this book, we'll be illustrating the steps in our process by presenting diagrams from an example project. Our example (developed by Sam Mancarella) will be a hardware/software system that most everyone is familiar with – an Audio Player.

The top level Package Diagram in Figure 9 shows how the example model is organized.



**Figure 9 – SysML models are organized into Requirements, Behavior, Structure, Constraints and Parametrics, and include both Hardware and Software Implementation models.**

You'll become intimately familiar with Sam's audio player example, as we'll be using it to illustrate the various activities on our roadmap throughout the following chapters.

# Chapter 2 – Audio Player Requirements

## Requirements Roadmap

Requirements are the foundation of a SysML model. The purpose of the system that you're modeling is to satisfy the requirements. So, as you'd expect, the roadmap begins with defining requirements (see Figure 1).

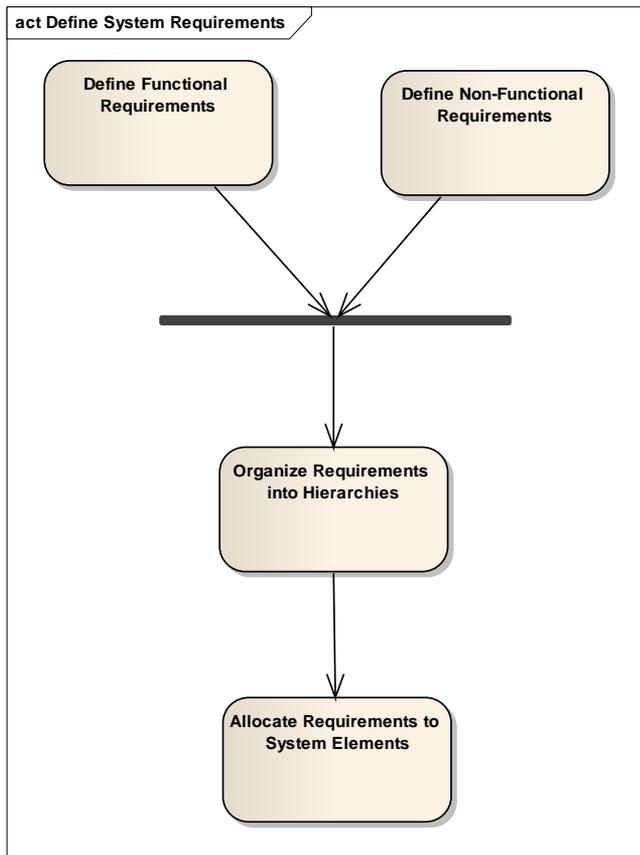


Figure 1 – Requirements Definition Roadmap

As you saw in Chapter 1, Requirements are usually classified as **Functional** (e.g. Requirements that represent specific system features or capabilities), and **Non-Functional** (e.g. Requirements that don't apply to specific features such as ease-of-use). It's important to organize these Requirements effectively, otherwise the Requirements model can become **Dysfunctional**<sup>5</sup>.

When you think about Requirements in a SysML model, you're considering Hardware Requirements, Software Requirements, and Requirements that relate to the Environment that your system will interact with. For example, our Audio Player will interact with its operating environment, which includes "listening conditions" (noise, weather), and the clothing of the listener.

---

<sup>5</sup> For more on avoiding Dysfunctional Requirements, see **Use Case Driven Object Modeling with UML – Theory and Practice**, by Doug Rosenberg and Matt Stephens.

These domain aspects drive downstream requirements which describe items such as shock resistance, waterproofing etc., because we expect the audio player to operate within the listeningDomain defined by this internal block diagram (ibd) which describes the listeningConditions to which the player will be subjected.

The 'blocks' shown in Figure 2 will be decomposed into their parts to describe this domain 'system'.

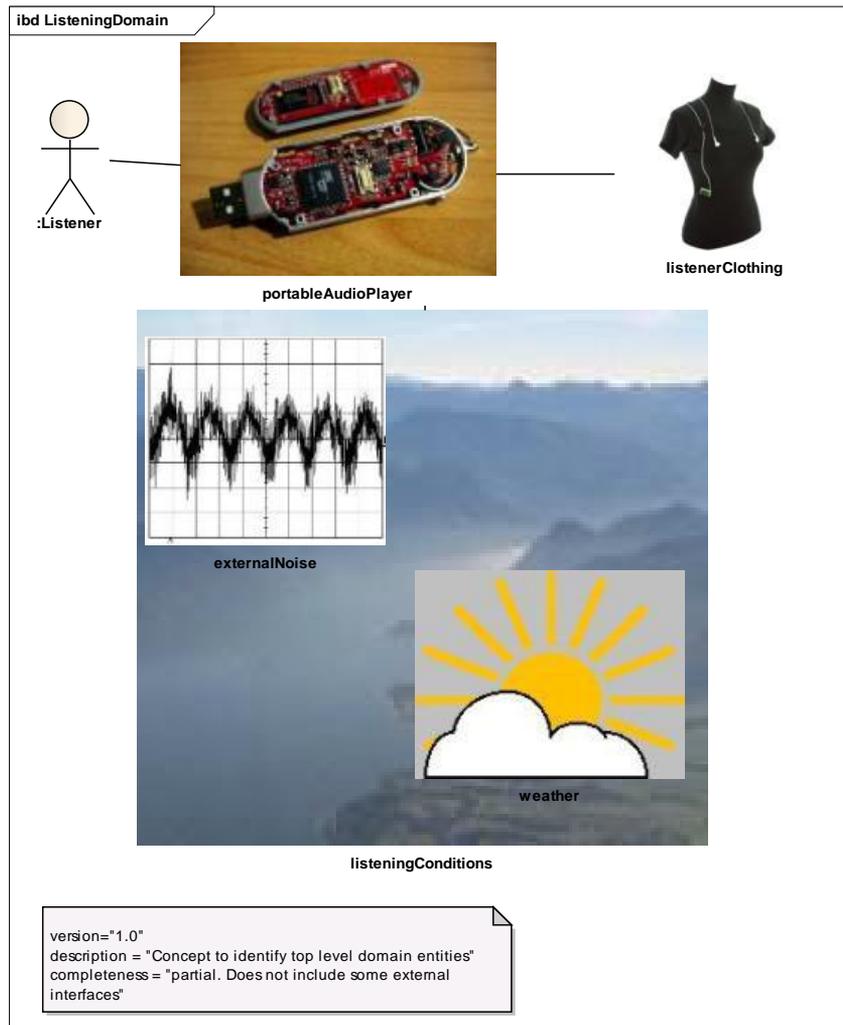


Figure 2 – SysML Models include Hardware, Software, and the Environment within which a system must operate.

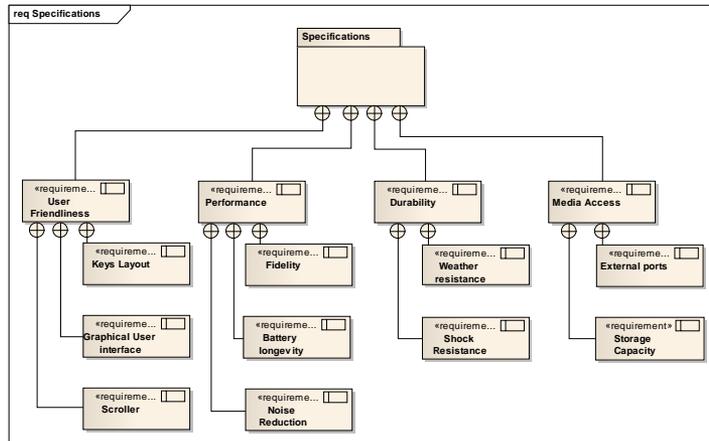
## Modeling Tip – It's easy to import graphics into Enterprise Architect Models

As you can see from the example in Figure 2, adding some graphics (photos, illustrations, etc.) to a model can make it far easier to understand. Enterprise Architect makes this easy to do. There are several ways to do this, but one of the easiest is to copy an image to the clipboard, then right-click an element on a diagram, select **Appearance** from the context menu, and then select **Apply Image from Clipboard**. It only takes a few seconds, but adds a lot to the readability of your model.

## Audio Player Requirements

SysML defines seven relationships between Requirements. These fall into two categories: relationships between Requirements, which include *containment*, *derive*, and *copy*; and relationships between requirements and other model elements, which include *satisfy*, *verify*, *refine*, and *trace*.

The “crosshair” notation in the Audio Player Requirements diagram below shows the use of *containment* to organize requirements hierarchically into Categories meaning that the Specifications Package OWNS the requirements, which in turn, own the ‘sub requirements’ beneath.



**Figure 3 – Requirements for our Audio Player are organized into Categories such as User Friendliness, Performance and Durability.**

Enterprise Architect has a couple of built-in features that make it easy to define which requirements are satisfied by which model elements, and to automatically generate a *relationship matrix* to show these relationships.

|                                  | Battery longevity | Durability | External ports | Fidelity | Graphical User interface | Keys Layout | Media Access | Noise Reduction | Performance | Scroller | Shock Resistance | Storage Capacity | User Friendliness | Weather resistance |
|----------------------------------|-------------------|------------|----------------|----------|--------------------------|-------------|--------------|-----------------|-------------|----------|------------------|------------------|-------------------|--------------------|
| Buttons                          |                   | ↑          |                |          | ↑                        |             |              |                 |             |          |                  |                  | ↑                 | ↑                  |
| Charging Unit - ADP2291          | ↑                 |            | ↑              |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Clothing                         |                   |            |                |          |                          |             |              |                 |             |          |                  | ↑                |                   |                    |
| Codec with Amplifier - TLV32...  |                   |            |                | ↑        |                          |             |              | ↑               | ↑           |          |                  |                  |                   |                    |
| Environment                      |                   |            |                |          |                          |             |              | ↑               |             |          |                  | ↑                |                   | ↑                  |
| Li-Ion Battery Monitoring Sys... | ↑                 |            |                |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| ListeningDomain                  |                   |            |                |          |                          |             |              | ↑               |             |          |                  |                  |                   |                    |
| Memory - MT42L32M64D2KH...       |                   |            |                |          |                          |             |              |                 |             |          |                  | ↑                |                   |                    |
| Noise                            |                   |            |                |          |                          |             |              | ↑               |             |          |                  |                  |                   |                    |
| Panasonic Li-Ion CGR18650AF      | ↑                 |            |                |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Portable Audio Player            |                   |            | ↑              |          |                          |             |              | ↑               |             |          | ↑                |                  |                   | ↑                  |
| PowerSubsystem                   | ↑                 |            |                |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Processing Subsystem             |                   |            |                |          |                          |             |              |                 | ↑           |          |                  |                  |                   |                    |
| Processor - TMS320VC507          |                   |            |                |          |                          |             |              |                 | ↑           |          |                  |                  |                   |                    |
| RS232                            |                   |            | ↑              |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Touch-screen - Toppoly TD0...    |                   |            |                | ↑        |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Transport Subsystem              | ↑                 |            | ↑              |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| USB - PL-2528                    |                   |            | ↑              |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| User Interface                   |                   |            |                |          | ↑                        |             |              |                 |             |          |                  |                  |                   |                    |
| Weather                          |                   |            |                |          |                          |             |              |                 |             |          |                  |                  |                   | ↑                  |

**Figure 4 – Enterprise Architect's Relationship Matrix makes it easy to see the allocation of Requirements to Blocks**

## Modeling Tip – allocate requirements to model elements using drag-and-drop

It's trivially easy to specify that a model element (such as a Block or a Use Case) satisfies a Requirement within an Enterprise Architect model. Simply drag the Requirement from the Project Browser on to the element which satisfies it. Enterprise Architect automatically establishes the link within the model.

It's also trivially easy to generate a matrix showing the allocations of Requirements to model elements using Enterprise Architect's Relationship Matrix.

The screenshot shows the 'Relationship Matrix' dialog box in Enterprise Architect. The 'Source' is 'Systems Engineering Model', 'Type' is 'UseCase', 'Link Type' is 'Realization', 'Target' is 'Systems Engineering Model', 'Type' is 'Requirement', and 'Direction' is 'Source -> Target'. The matrix below shows the following allocations:

| Use Case                       | Battery longevity | Durability | External ports | Fidelity | Graphical User interface | Keys Layout | Media Access | Noise Reduction | Performance | Scroller | Shock Resistance | Storage Capacity | User Friendliness | Weather resistance |
|--------------------------------|-------------------|------------|----------------|----------|--------------------------|-------------|--------------|-----------------|-------------|----------|------------------|------------------|-------------------|--------------------|
| Adjust Volume                  |                   |            |                |          |                          |             |              | ↑               |             |          |                  |                  |                   | ↑                  |
| Charge Battery                 | ↑                 |            |                |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Connect To Computer            |                   |            | ↑              |          |                          |             | ↑            |                 | ↑           |          |                  |                  |                   | ↑                  |
| Copy track from external media |                   |            |                |          |                          |             |              |                 | ↑           |          |                  |                  | ↑                 |                    |
| Create Playlist                | ↑                 |            |                |          | ↑                        | ↑           |              |                 | ↑           |          |                  |                  | ↑                 |                    |
| Download track                 | ↑                 | ↑          | ↑              |          |                          |             |              |                 | ↑           |          |                  |                  | ↑                 |                    |
| Listen Audio                   |                   |            |                | ↑        |                          | ↑           |              | ↑               | ↑           |          |                  |                  |                   | ↑                  |
| Maintain Audio Player          |                   |            |                |          |                          |             | ↑            |                 |             |          |                  |                  |                   |                    |
| Maintain Playlist              |                   |            |                |          |                          |             | ↑            |                 |             |          |                  |                  |                   |                    |
| Operate Audio Player           |                   | ↑          |                |          | ↑                        | ↑           |              | ↑               | ↑           | ↑        |                  |                  |                   | ↑                  |
| Pause                          |                   |            |                |          | ↑                        |             |              | ↑               |             |          |                  |                  |                   | ↑                  |
| Play                           |                   |            | ↑              | ↑        |                          |             |              | ↑               | ↑           |          |                  |                  |                   | ↑                  |
| Power On                       | ↑                 |            |                |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| Record Audio                   |                   |            |                | ↑        |                          |             |              | ↑               | ↑           |          |                  |                  |                   | ↑                  |
| Replace Battery                | ↑                 | ↑          |                |          |                          |             |              |                 |             |          | ↑                |                  | ↑                 |                    |
| Replace Headphones             |                   |            | ↑              |          |                          |             |              |                 |             |          |                  |                  | ↑                 |                    |
| Replace Skin                   |                   | ↑          |                |          | ↑                        | ↑           |              |                 |             |          | ↑                |                  | ↑                 | ↑                  |
| Stop                           |                   |            |                |          | ↑                        |             |              |                 | ↑           |          |                  |                  |                   | ↑                  |
| Top Level                      |                   |            |                |          |                          |             |              |                 |             |          |                  |                  |                   |                    |
| View Existing Playlist         |                   |            |                |          |                          |             | ↑            |                 |             |          |                  | ↑                |                   |                    |
| View New Tracks                |                   |            |                |          |                          |             | ↑            |                 |             |          |                  | ↑                |                   |                    |

Figure 5 – Enterprise Architect's Relationship Matrix makes it easy to see the allocation of Requirements to Use Cases

## Modeling Tip – Use Enterprise Architect's Relationship Matrix to show which model elements satisfy which Requirements

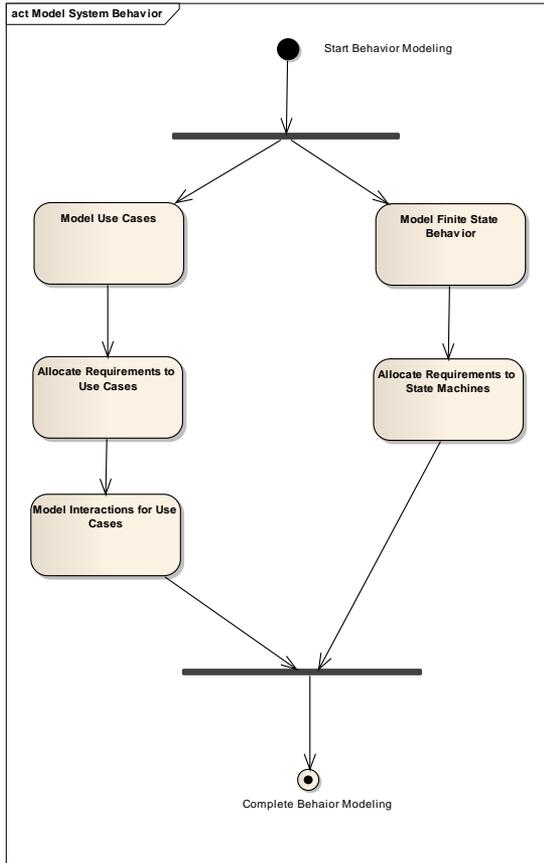
Once the allocation relationships have been specified using drag-and-drop, or by using the "Require" tab on the Specification dialog of all Enterprise Architect elements, you can generate a cross-reference table of Requirements against any type of model element by selecting **Relationship Matrix** from the View menu. Simply specify the scope (desired Package) of your search using the Source and Target buttons, and the type of elements you wish to cross-reference, and Enterprise Architect does the rest. The Matrix can be exported to a Comma Separated Value (CSV) file using the Options button.

In the upcoming chapters, you'll see how the Requirements we've identified here are satisfied by various aspects of the Audio Player SysML model.

# Chapter 3 – Audio Player Behavior

## ***Behavior Modeling Roadmap***

Behavior Modeling describes the dynamic behavior of the System as it interacts with users and with the environment. You'll use interaction diagrams and use cases to model interactions between users and the system, and state machines to describe event-driven behavior that's not user-centric. Figure 1 shows the Roadmap activities.



**Figure 1 – Behavior Modeling Roadmap**

As you can see, you approach behavior modeling in two parallel branches, one for use cases and the other for state machines. Each branch includes allocation of Requirements to model elements (use cases or states). Use cases are described in natural language at the high level, and are detailed on interaction (sequence) diagrams.

We'll follow the roadmap through the remainder of this chapter by exploring the dynamic behavior of our Audio Player example. Then in Chapter 4 we'll explore the system structure that supports the desired behavior. We use the terms "static" and "dynamic" to describe the structural and behavioral parts of the model; structure is *static* in that it doesn't change once it's defined, while behavior is *dynamic* – changing based on user actions or external events.

## Audio Player Behavior Model

Here we can see the two branches of the dynamic model for our Audio Player. User-centric scenarios, such as Operating the Audio Player, are modeled with use cases, while we can model the Operating States of the device with state machines. Note that Playlist Maintenance has a use case description and is also described using a state machine. Whatever diagrams help to tell the story can be used.

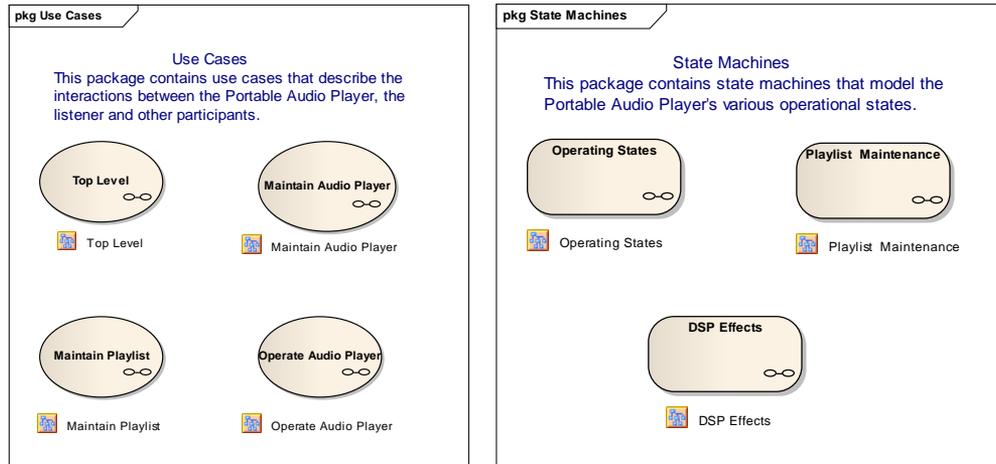


Figure 2 – Behavioral models include Use Cases, Interactions, and State Machines

### Modeling Tip – Models should tell a story

A model's primary purpose is to serve as a communication vehicle to promote a shared understanding about the system being modeled between stakeholders, end-users, analysts, designers, software and hardware engineers, and quality assurance personnel. Always optimize models for readability, and make sure you “tell the story” clearly and unambiguously.

Here are the Top Level use cases for the Audio Player. The “eyeglass” icon on the use case bubbles indicate that a child diagram exists, showing more detail about the use case.

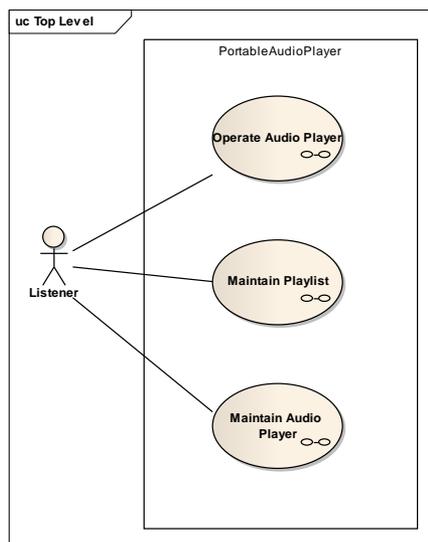
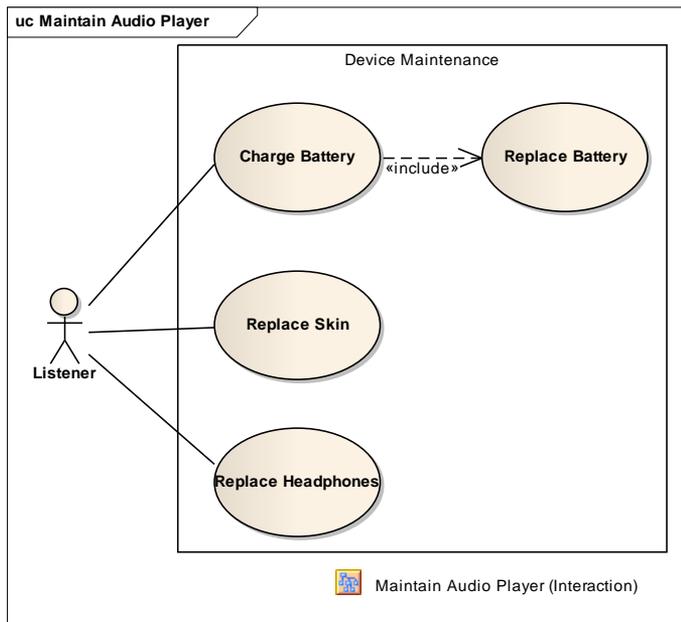


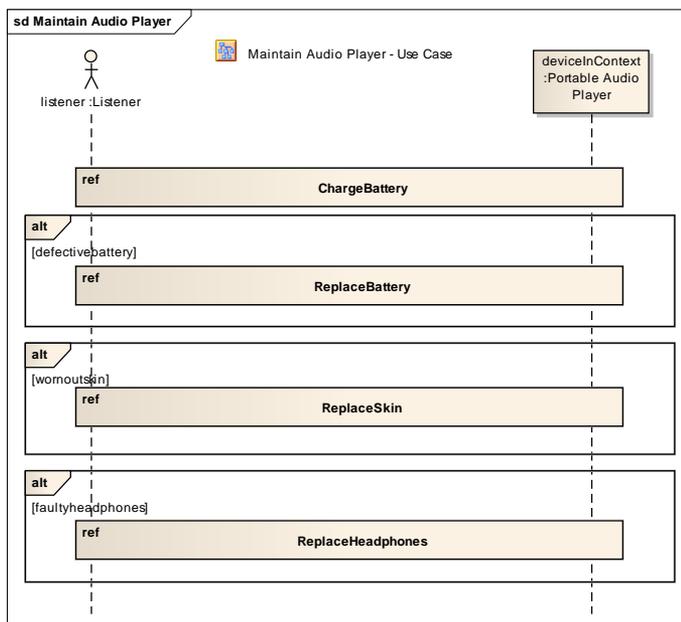
Figure 3 – Audio Player Top Level Use Cases

Enterprise Architect makes it easy to “drill down” to a child diagram for composite elements. Here’s a child use case diagram for audio player maintenance.



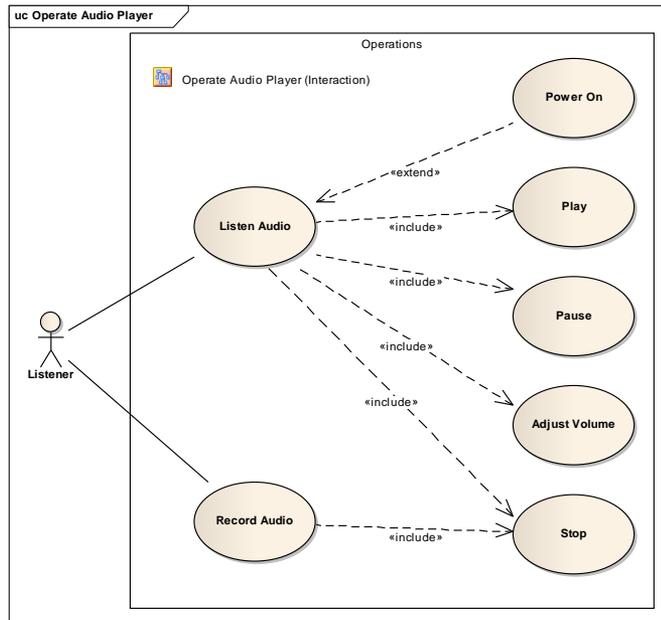
**Figure 4 – Use cases for maintaining the audio player hardware include charging and replacing the battery, and replacing the skin and the headphones.**

Enterprise Architect supports “diagram references” for hyperlinking one diagram to another. You can see the reference to the interaction diagram (Figure 5) on the diagram above, and a link back to the use case view on that diagram.



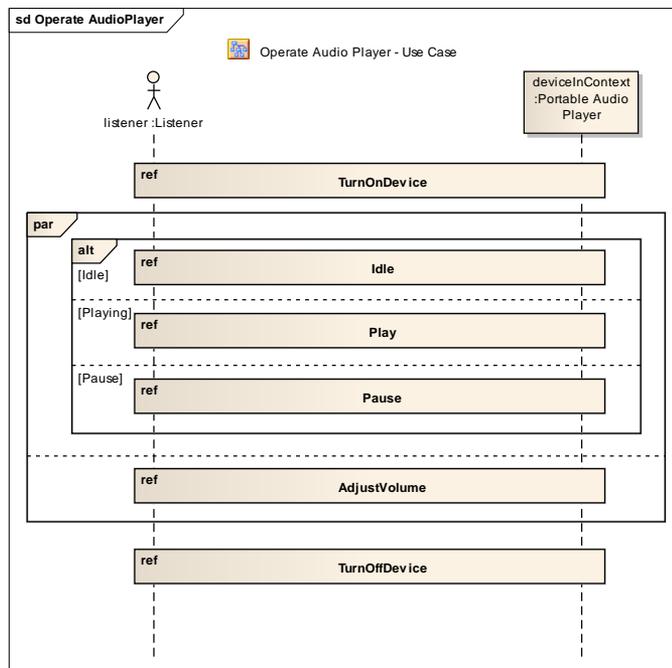
**Figure 5 – The interaction diagram for maintaining the audio player shows 3 alternate courses of action (defective batteries, worn out skin, and faulty headphones) and one normal course (charging the battery).**

Here are the use cases for the basic operations of the Audio Player.



**Figure 6 – Audio Player Use Cases for Listening and Recording**

As in the Maintenance use cases, the use case diagram and interaction diagram (Figure 7) are cross-linked using Enterprise Architect diagram references. This diagram shows that the Idle, Play, Pause, and Adjust Volume paths can all be performed in parallel.



**Figure 7 – Audio Player Interaction diagram for Listening/Recording**

Use cases describe how a user interacts with the system. In the next section, you'll see how to describe event-driven behavior using state machines.

## Audio Player State Model

For embedded systems, it's often advantageous to describe behavior in terms of operating states, triggering events and system actions. SysML (identically to UML) uses state charts to describe these finite state aspects of a system.

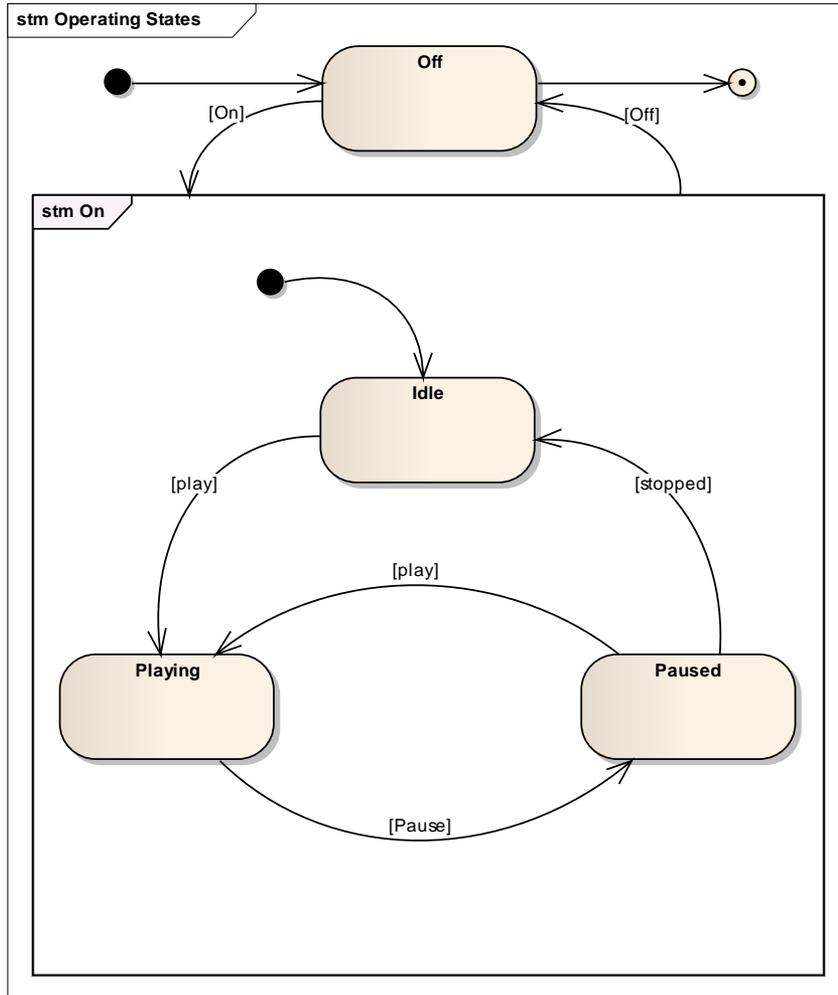


Figure 8 – Audio Player Operating States

State charts allow nesting of substates on a single diagram. Figure 8 shows the detailed behavior of the “On” state of the audio player on the same diagram that shows the “On/Off” behavior. To allocate Requirements to states, simply drag the Requirement from the Enterprise Architect Project Browser and drop it onto the state bubble.

State machines relate operating states of a system (or block) to *triggering events* such as pressing a button. Figure 9 shows how toggling the “Audio EQ” button causes the system to cycle between various audio effects.

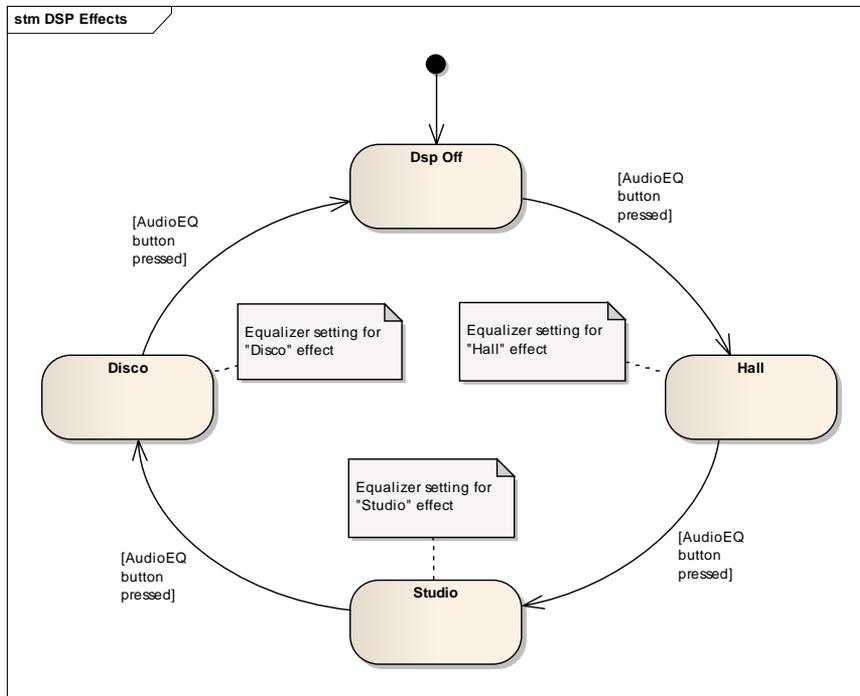


Figure 9 – State Machine for Digital Signal Processing Audio Effects

There's no absolute rule for choosing when to “tell the story” with use cases and when to use state diagrams. The best guideline is to simply use whichever diagram that tells the story best. Sometimes, the best choice is to use both.

## Combining Use Cases and State Machines

Here's an example that shows how use cases, interaction diagrams, and state machines can all be used to describe different aspects of how our audio player system operates.

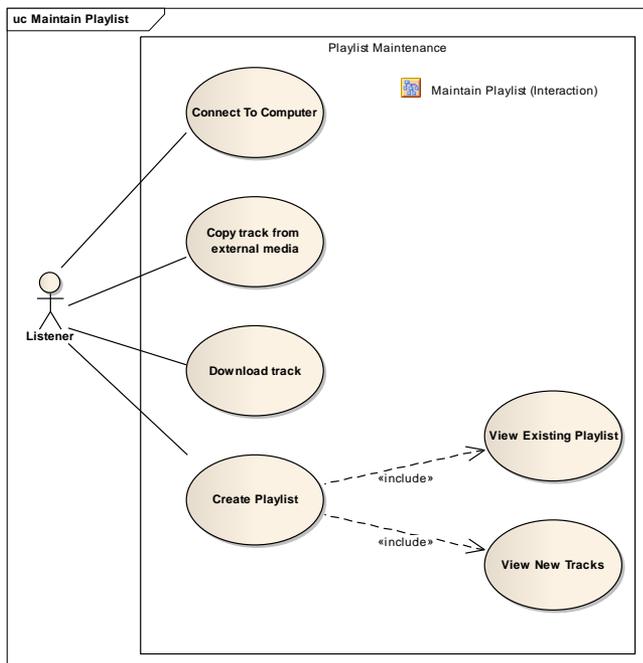
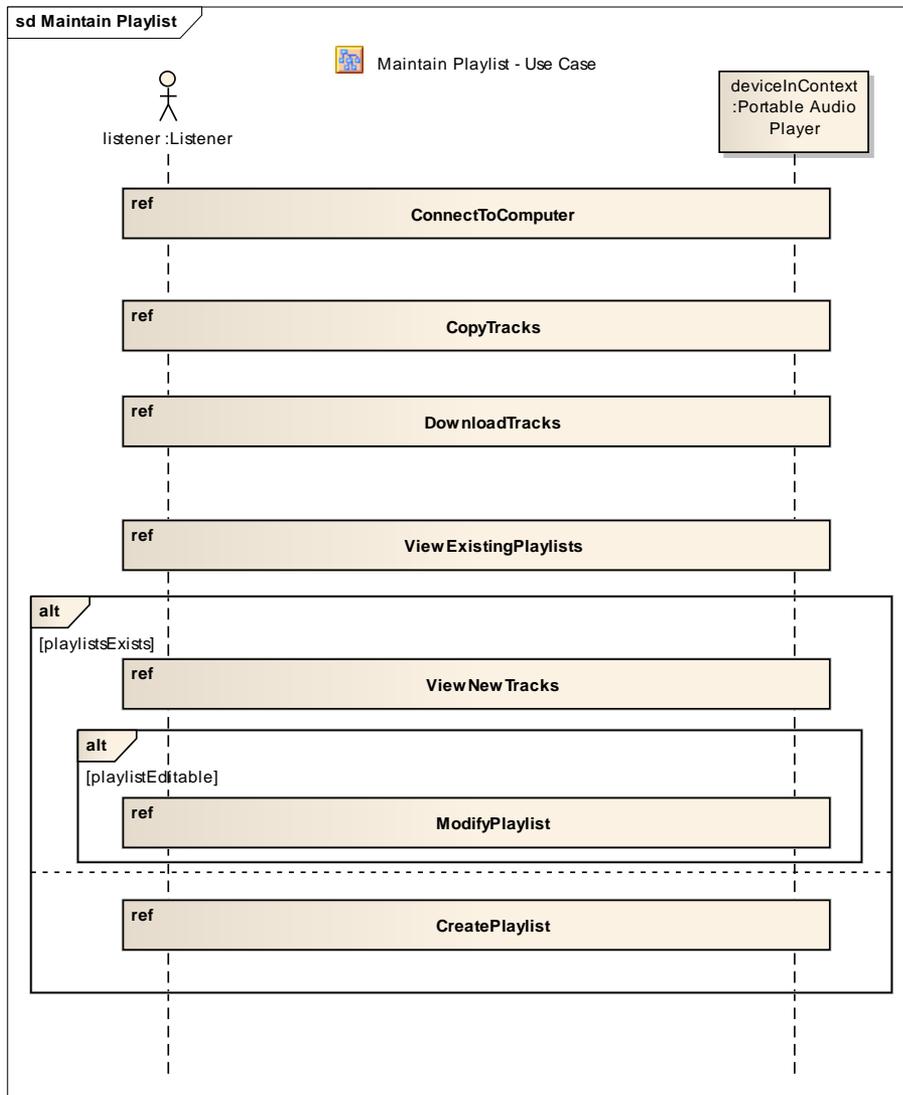


Figure 10 – Use Case Diagram for Playlist Maintenance

Each diagram provides a different perspective on the system we're modeling. We can use as many views as necessary to "tell the story" so that there are no misunderstandings as we progress from defining Requirements through hardware and software development. Figure 11 shows the various scenarios for maintaining playlists, while Figure 12 takes a more event-driven perspective.

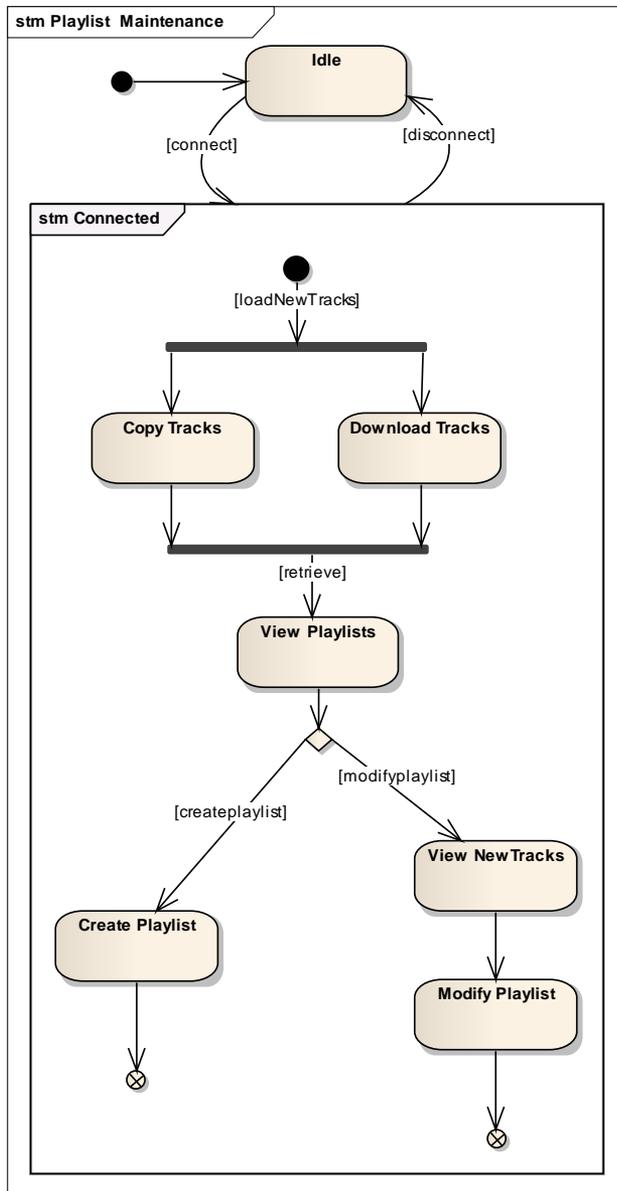


**Figure 11 – Scenarios for maintaining playlists**

Note that in order to Modify a Playlist, the Playlist must already exist and be editable. However, tracks may be downloaded and copied independently of those conditions.

The state machine shown in Figure 12 provides a different perspective. The top level state machine shows how the behavior depends on connecting/disconnecting the audio player to/from the music server. As you can see, all of the real behavior of maintaining playlists happens when the device is connected.

An activity diagram is used to detail the behavior of the audio player when it's connected. Forks and joins (the solid black horizontal lines) on the activity diagram are used to show parallel paths.



**Figure 12 – State/Event behavior for Playlist maintenance**

The combination of use cases, interaction diagrams, state charts, and activity diagrams allow you to specify the dynamic behavior of the system in great detail. Additionally, you can allocate Requirements to use cases, states, activities, and other model elements.

This wraps up our discussion of Behavior Modeling, as we've completed all the steps in the Roadmap. In the next chapter we'll explore the Roadmap for defining system structure using blocks.

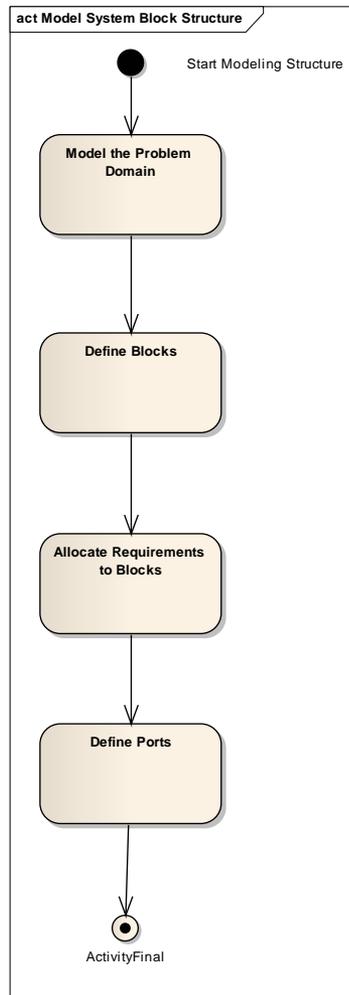
## Chapter 4

# Audio Player Structure

Now that you've looked at Requirements and Behavior Modeling, it's time to explore how you can use SysML and Enterprise Architect to describe the Structure of a system. As usual, we'll illustrate by describing the structure of our Audio Player example.

### ***Roadmap: Define Structure***

Figure 1 shows our Roadmap for modeling Structure.



**Figure 1: Roadmap - Model Structure**

In SysML, the Block is the primary unit used to describe Structure. Blocks can represent hardware or software elements. Blocks can have Ports, which represent the inputs to, and outputs from, the Block.

## Modeling the Problem Domain

Our Structural modeling roadmap starts with a familiar step to anyone who has seen ICONIX Process for Software- or Business-Domain Modeling. When you build a domain model, you define a set of abstractions based upon things in the real world. Figure 2 shows a domain model for our Audio Player. As you can see, the domain model can include real-world elements that are external to our system, such as Clothing and the surrounding Environment.

The purpose of the Problem Domain model is to describe the 'System' in which our Audio Player will operate under. It's a 'system' model used to describe the 'context' of our device design – from Requirements through to implementation. Figure 2 shows which systems will interact together with our audio player in a concept known as a 'System of Systems' design.

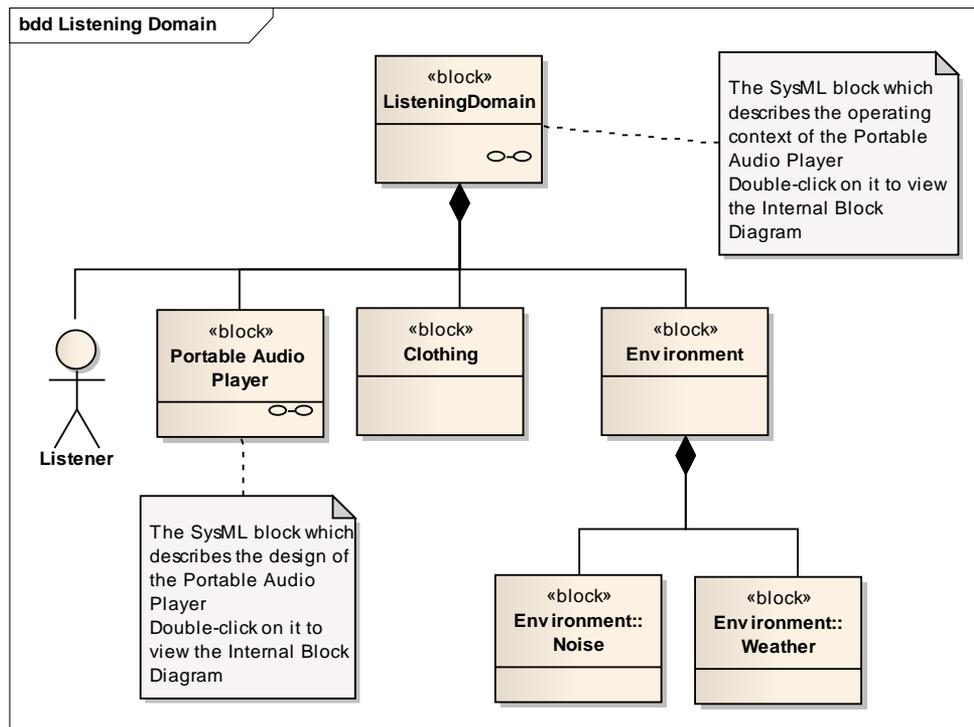


Figure 2 – Audio Player Domain Model

The "black/filled diamond" association in Figure 2 represents a composition relationship, indicating (for example) that the Environment is "composed of" Noise and Weather.

There are two levels of Block diagramming in SysML: Block Definition Diagrams (BDDs), and Internal Block Diagrams (IBDs). We'll explore these in order in the next few sections of this chapter.

## Modeling Block Structure (Block Definition Diagrams)

Figure 3 shows the "child" block definition diagram that details the high-level structure of our Audio Player. The purpose of the BDD is to describe the composition of a block by relating nested blocks to each other using the composition relationship.

As you can see, the Audio Player is composed of four main subsystems; Power, Processing, User Interface, and Transport. Each of these is modeled as a block and further decomposed into sub-blocks.

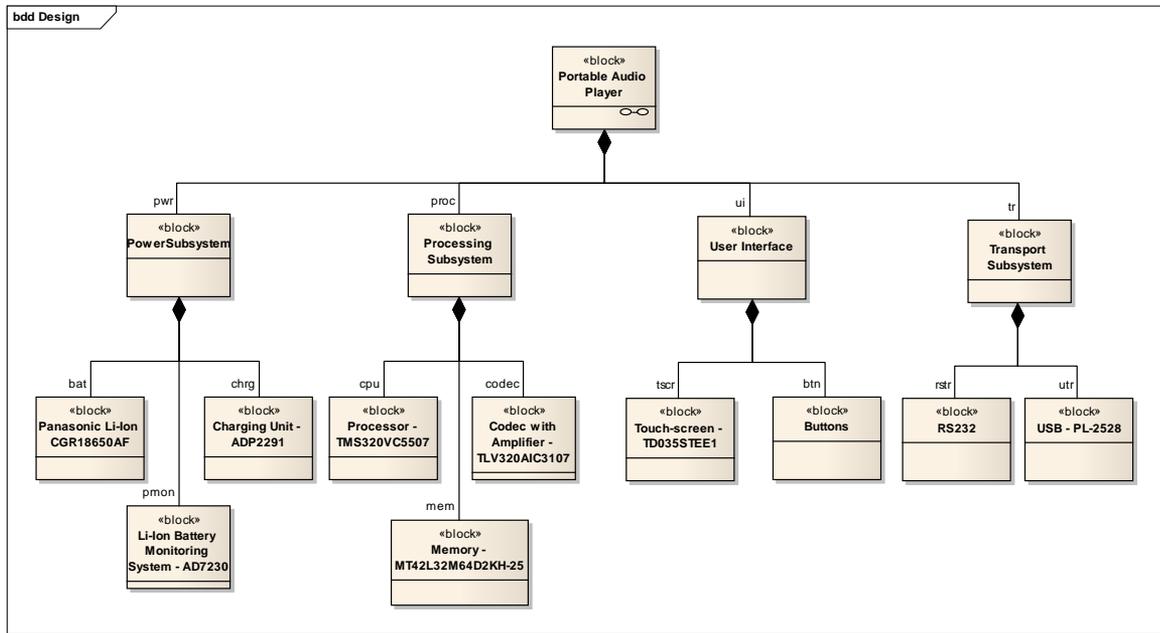


Figure 3 – Audio Player Block Structure

Figure 4 shows the details of the Power Subsystem. It's composed of a Lithium-Ion Battery, a Charging Unit, and a Monitoring System. Each of these blocks has a port which represents the electric current that operates the Audio Player.

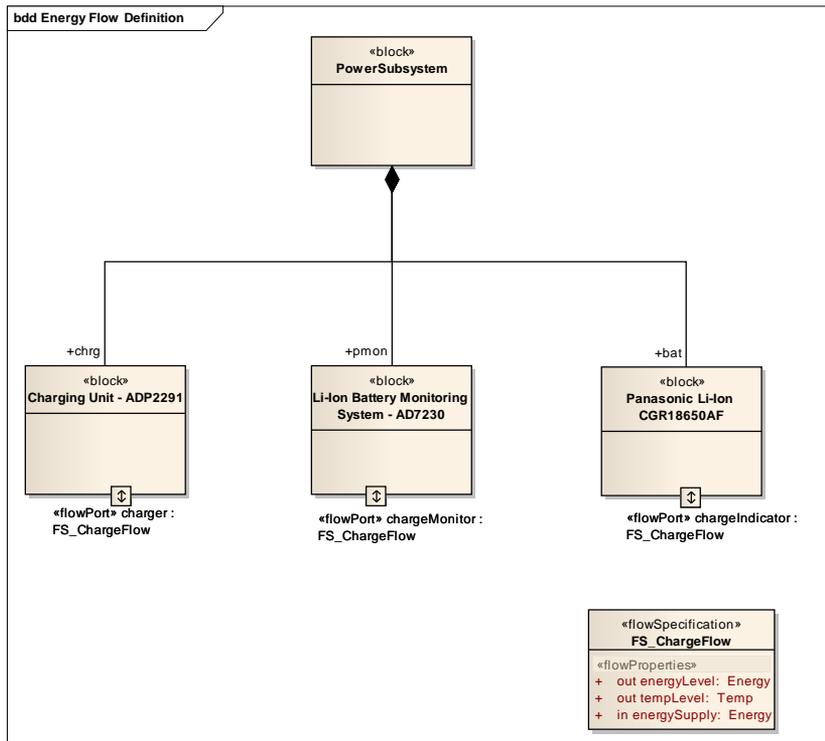


Figure 4 – Audio Player Power Subsystem

## Modeling Block Internals (Internal Block Diagrams)

Figure 5 shows a simple Internal Block Diagram (IBD) for the Power Subsystem. The purpose of the IBD is to describe in detail how each of the block 'parts' are connected together to form the subsystem in question. The IBD describes the 'what and how' of the block composition. Each of the Parts represents a composition relationship in the corresponding BDD in the previous section.

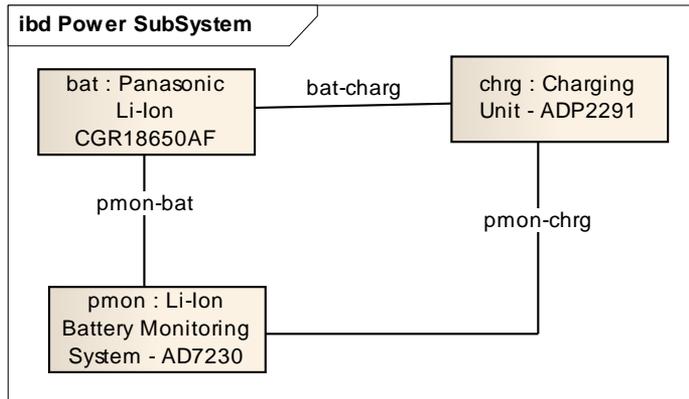


Figure 5 – Power Subsystem IBD

The Internal Block Diagram specifies the connection of Parts within a Block. As you can see in Figure 6, it's possible to show multiple levels of nesting on a single IBD.

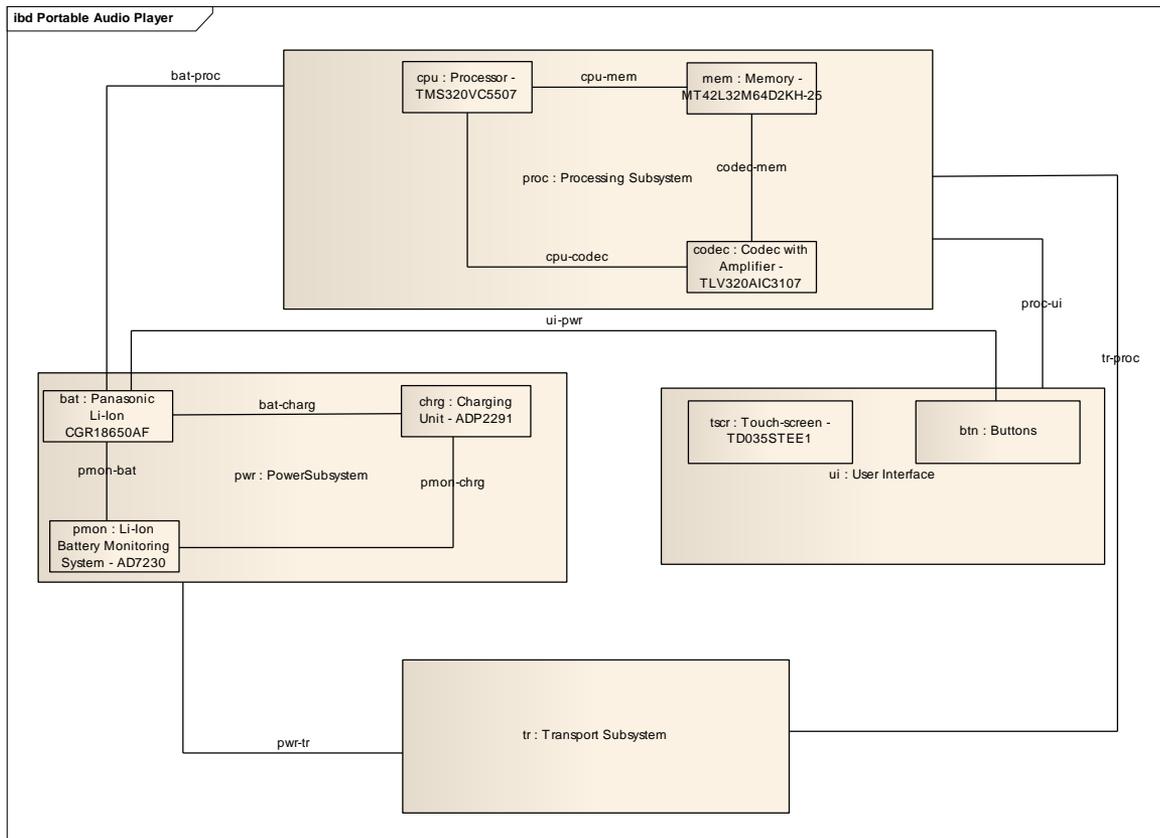


Figure 6 – Multi-level IBD showing interconnection of parts for the Audio Player

Figure 7 shows the internals of the Processing Subsystem. As you can see, the CPU connects to a Memory Unit and a Codec/Amplifier.

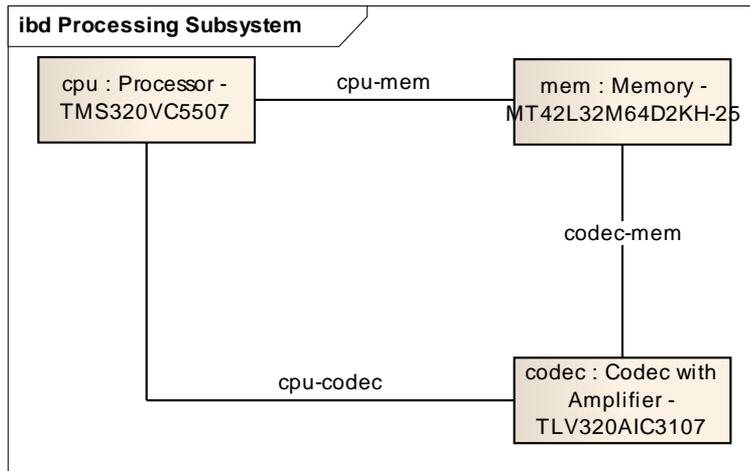


Figure 7 – Audio Player Processing Subsystem Block Internals

## Define Ports

The final step in our Roadmap for Modeling Structure is to define the Ports. Figure 8 illustrates data flow between the User Interface, Processing Subsystem, Transport Subsystem, and the USB and RS-232 connectors on the Audio Player.

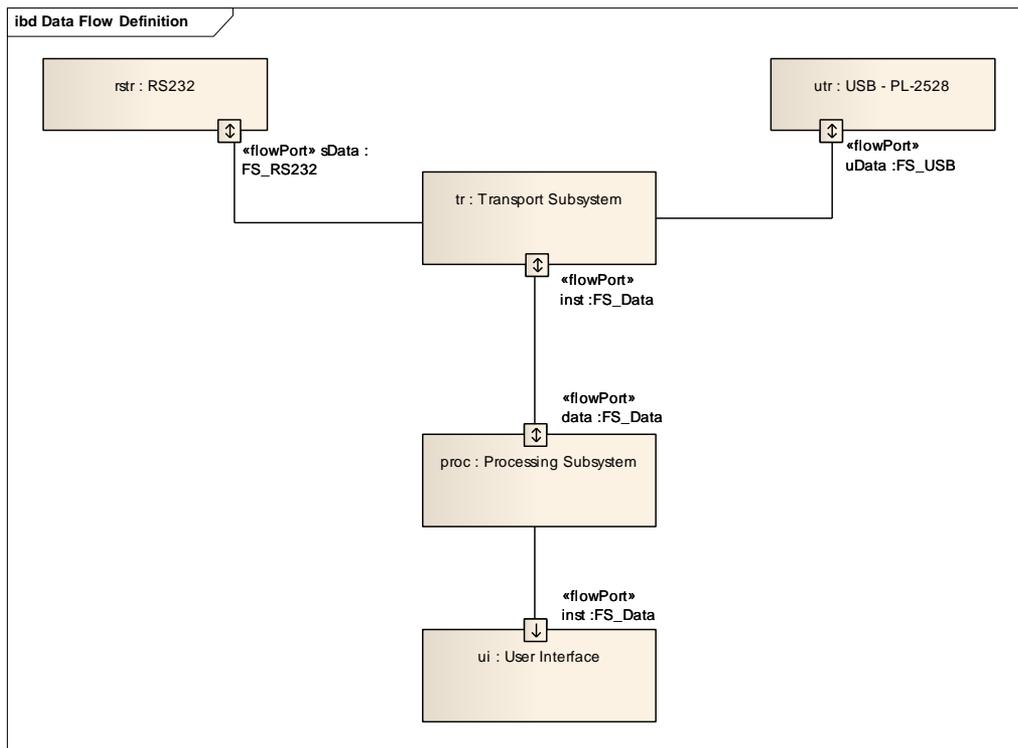


Figure 8 – Audio Player Dataflow between Subsystems

Figure 8 illustrates a type of Port called a flowPort. The SysML flowPort is typed by a FlowSpecification which describes the properties and behavior associated with the port.

A flowPort describes also the directionality of the items flowing through it (in/out/conjugate) SysML also includes standardPorts, which can either provide an interface or require an interface. ItemFlows on the connectors (the arrows) describes what is flowing across the connections and through the ports. In the example above, it is Data which flows through these connections.

## Audio Player Hardware Components

Finally, Figure 9 shows the hardware components of our Audio Player.

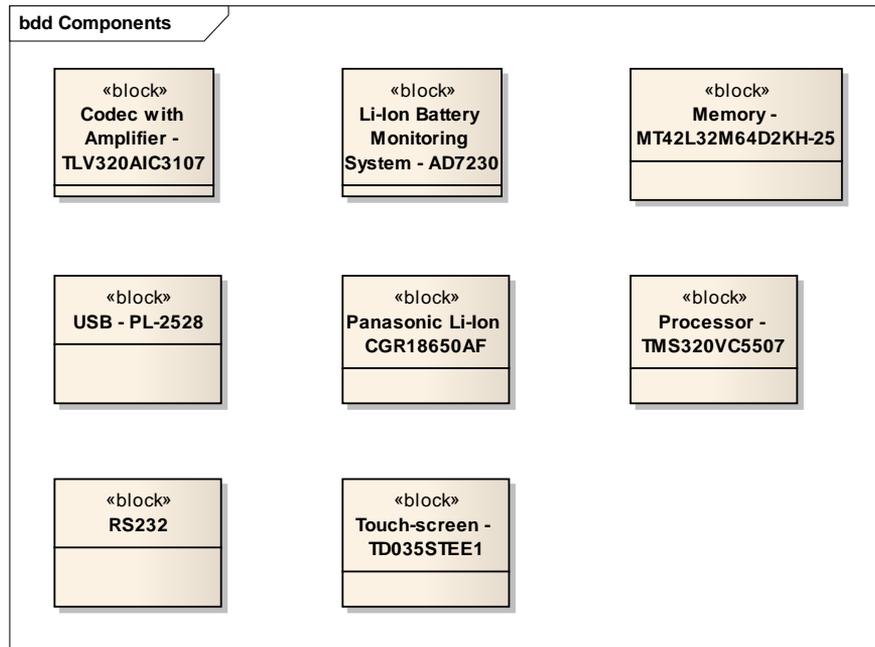


Figure 9 – Hardware Components are modeled as Blocks

Modeling the hardware components as blocks makes it possible to allocate requirements to hardware. Enterprise Architect makes it easy to allocate requirements to any of the model elements discussed in this chapter.

This concludes our discussion of Blocks, Parts, and Ports. We've built the foundation for our SysML model over the last 3 chapters where we covered Requirements, Behavior Modeling, and Structural Modeling. The last 3 chapters of the book introduce more advanced aspects of SysML and powerful capabilities of Enterprise Architect System Engineering Edition. In the next chapter we'll introduce Constraints and Parametrics, and then proceed to hardware and software implementation.

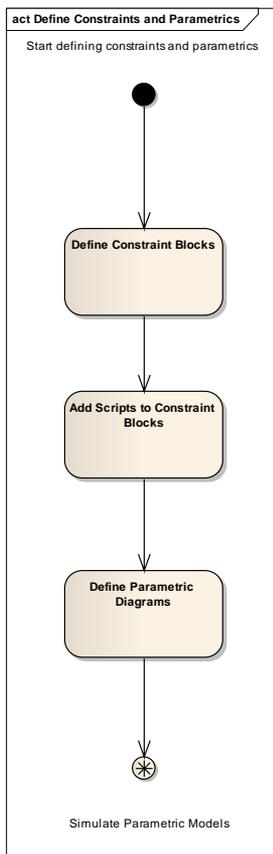
## Chapter 5

# Audio Player Constraints and Parametrics

One of the biggest differences between SysML and UML is the ability to simulate portions of a SysML model, based on mathematical and physical laws that describe key aspects of the system. One of the biggest differences between Enterprise Architect Systems Engineering Edition and other SysML modeling tools is *Enterprise Architect's ability to do that simulation within the modeling tool*, as opposed to simply interfacing to external simulators. We'll explore these capabilities in this chapter, starting, as usual, with our process roadmap.

### ***Constraints and Parametrics Roadmap***

Our constraints and parametrics roadmap has two sections. The first step, detailed in Figure 1, is to define the Constraints and Parametrics. The second step, shown in Figure 2, is to configure and run the Simulation. This entire process can be done completely within the Enterprise Architect Systems Engineering Edition – speeding convergence towards an engineering solution that meets the Requirements. We'll spend the remainder of this chapter following the steps in our roadmap for the Audio Player.

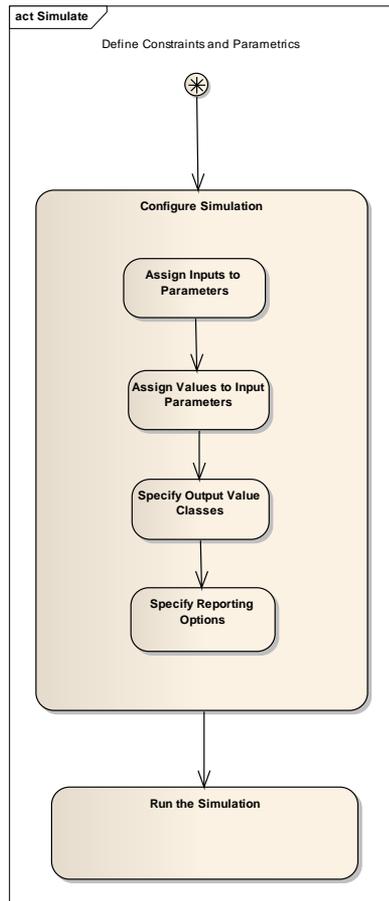


**Figure 1 – Roadmap: Define Constraints and Parametrics**

SysML parametric models support the engineering analysis of critical system parameters, including the evaluation of key metrics such as performance, reliability and other physical characteristics. They unite requirements models with system design models by capturing executable constraints based on complex mathematical relationships. In SysML, parametric

models can also be used to describe the actual requirements themselves (e.g. “The internal combustion engine shall deliver its torque in accordance with the ‘attached’ parametric characteristics.” The parametric can describe the ‘graph’ used to describe the torque curve for the engine).

As you can see in Figure 1, defining parametric models using Enterprise Architect’s System Engineering Edition involves defining Constraint Blocks, Adding Scripts to the Constraint Blocks, and Defining Parametric Diagrams. Once the parametric models are defined, they can be simulated, as shown in Figure 2.



**Figure 2 – Roadmap: Configure and Run Simulation**

Simulating a SysML parametric model is simply a matter of configuring the simulation, and then running it. Having the ability to do all of this within Enterprise Architect makes it much faster and easier to make engineering tradeoffs in the model without having to break away from Enterprise Architect into another tool, and tightens the engineering feedback loop, making it much faster to converge on a solution that meets your project’s Requirements.

## Define Constraint Blocks

To build a parametric model, you create a collection of SysML Constraint Blocks that formally describe the function of a constraint in a simulation model. Each Constraint Block contains properties that describe its input and output parameters, as well as a Script that describes the constraint's executable component. Figure 3 shows constraint blocks for some of the underlying mathematical functions that make our Audio Player work.

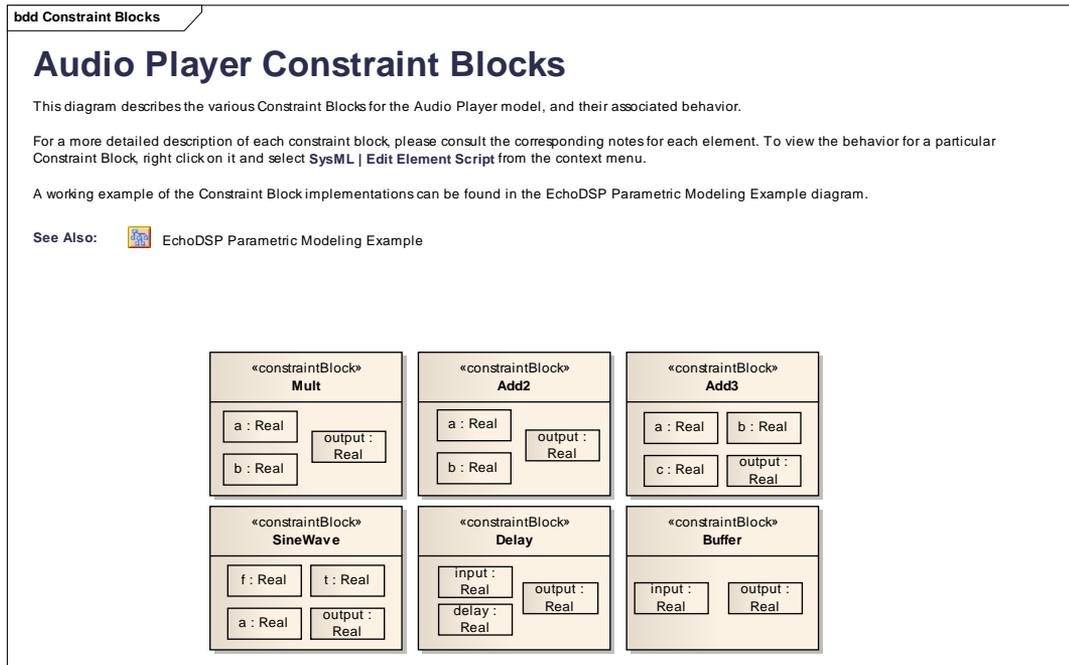


Figure 3 – Constraint Blocks for Audio Player functions

Next, create a SysML Constraint Block to contain the Parametric model you wish to simulate. In Figure 4 we're going to simulate the Echo Digital Signal Processing (DSP) function.

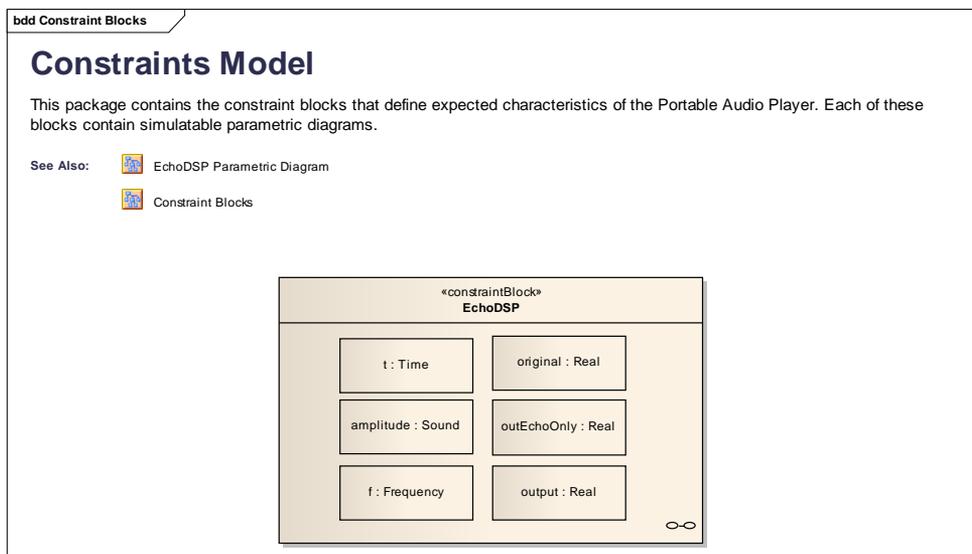


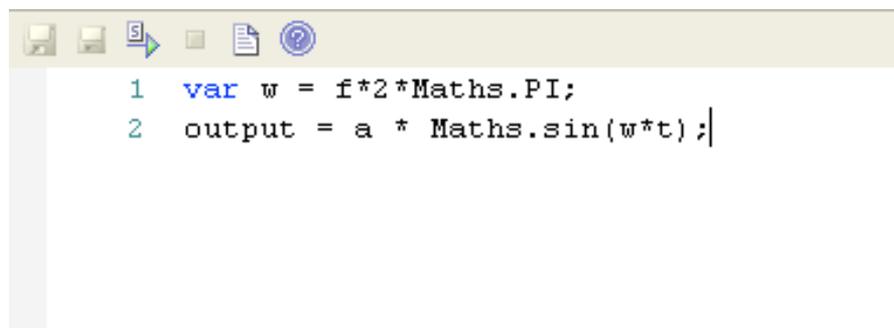
Figure 4 – Constraint Block for the Echo DSP function

As you can see in Figure 4, our Echo function takes an original signal which is a sine wave amplitude and frequency, and delays that signal by some amount of time to produce an echo, then can output either the echo signal only, or a composite signal that adds the echo to the original signal. To do this, we'll make use of the "primitive" constraint blocks shown in Figure 3 for Sinewave, Delay, Add, etc.

## **Add Scripts to Constraint Blocks**

Once your constraint blocks have been created, it's time to add Scripts. This is where you express the relationship / behavior of the constraint block as an executable script. In Enterprise Architect, right-click on each of the Constraint Blocks and select the *SysML | Add Element Script* context menu option to add a script to the constraint block.

Figure 5 shows a script for the SineWave constraint block. Similar scripts exist for the Buffer, Delay, Add, and other constraint blocks.



```
1 var w = f*2*Maths.PI;
2 output = a * Maths.sin(w*t);|
```

**Figure 5 – Script for the SineWave Constraint Block**

Attaching scripts to constraint blocks provides the underlying mathematical foundation for running simulations. The precise behavior of each block is specified in equation form, using the inputs and outputs by name where appropriate, thus allowing the simulation to take place.

### **Modeling Tip: Enterprise Architect supports scripting in several languages**

Scripts can be written in either JavaScript, Jscript or VBScript, and the user can use any other assemblies, components, or APIs in their constraint block script.

Note that simulating a constraint block requires the script across all constraint blocks to be written in the same language.

## Define Parametric Diagrams

The Parametric model contains properties and occurrences of constraint blocks as *Constraint Property* elements, connected in a Parametric Diagram.

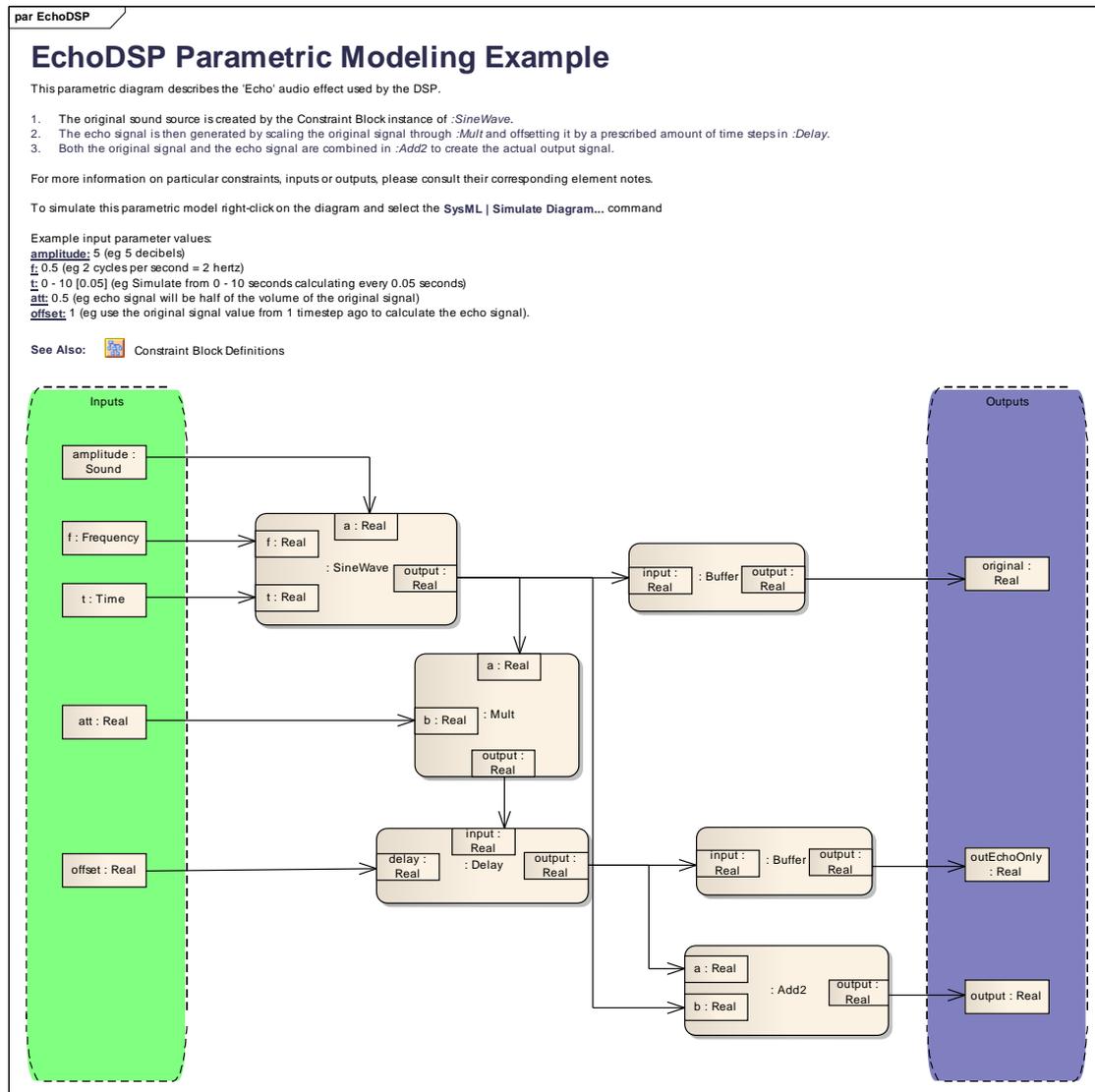


Figure 6 – Parametric Diagram for Echo DSP

The parametric diagram connects instances of primitive constraint blocks together to transform a set of inputs into a set of outputs. In Figure 6, we're taking an input SineWave, delaying and attenuating it, then adding that signal to the original input SineWave to produce an Echo effect. You can adjust parameters like attenuation and offset, and simulate, until you've produced the desired effect.

This brings us to the second portion of our roadmap, Configuring and Executing the Simulation.

## Configure Simulation

Now that your constraint blocks, scripts, and parametrics have been defined, you're ready to simulate, so let's right-click within a Parametric Diagram and select the *SysML | Simulate Diagram...* context menu option. The Simulation Configuration dialog displays (see Figure 7).

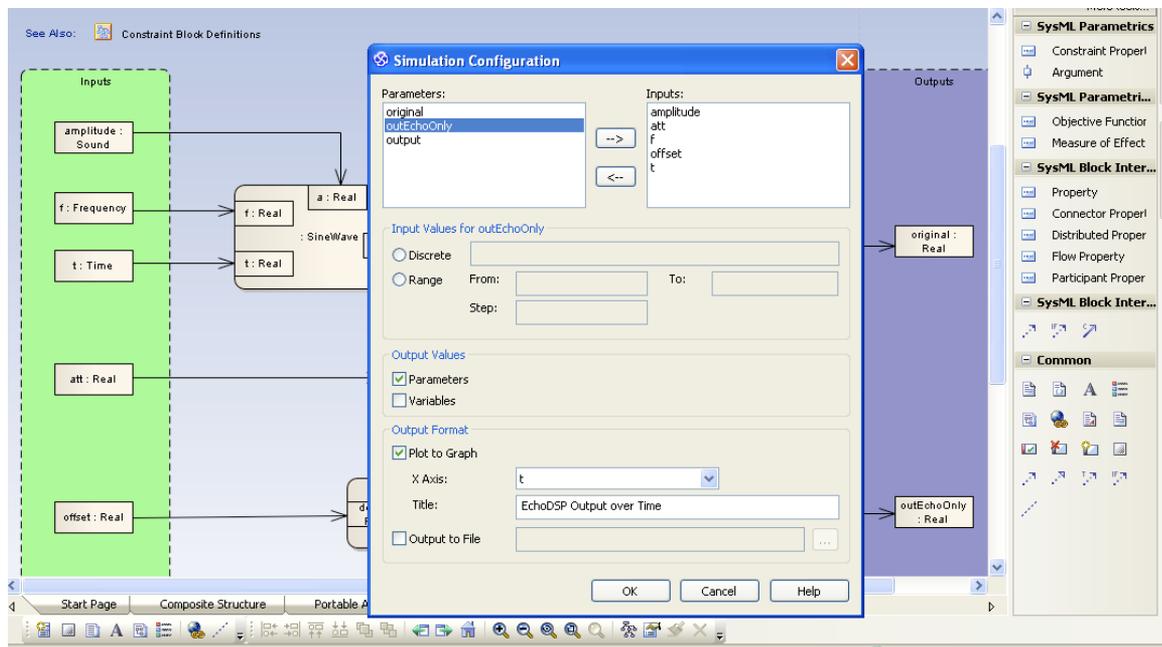


Figure 7 – Configuring the Simulation

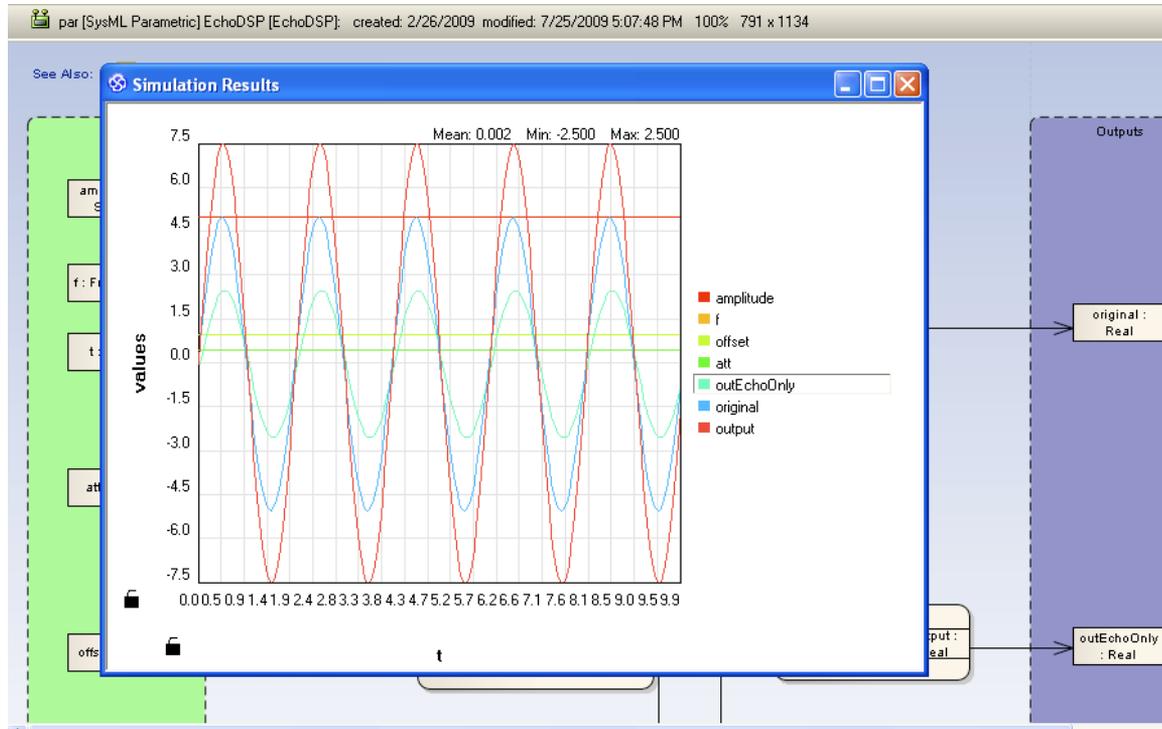
Fill out the Simulation Configuration dialog as follows:

- **Assign Inputs.** The Parameters panel lists all of the parameters that can be assigned input. Select each of the required parameters and click on the right arrow button to assign them as input. Parameters designated as input parameters are listed in the Inputs panel on the right. There must be at least one input parameter assigned for the simulation to execute.
- **Assign a set of values for each of the designated input parameters.** For each input parameter, in the Input Values panel select one of the two possible value kinds: Discrete or Range.
- **Specify the classes of output value: Parameters or Variables.**
- **Specify how the simulation results are to be reported.** The Output Format panel enables you to choose how the simulation outputs the simulation data. Depending on your configuration selections, the simulation's results are either written to a comma-separated CSV file or graphed in a 2-dimensional plot.

Once you've completed configuring your simulation, you're just about done.

## Run the Simulation

To simulate your SysML model, click on the **OK** button to execute.



**Figure 8 – Simulation results can be displayed directly within Enterprise Architect.**

While there is an option to export the simulation results to a CSV file, the ability to display simulation results directly within Enterprise Architect is one of the features that sets it apart from other SysML modeling tools. Having everything in a single tool makes it quick and easy to tweak design parameters so that your system meets its required performance targets.

In the next two chapters, we'll explore how Enterprise Architect helps to transform a SysML model into both hardware and software solutions.

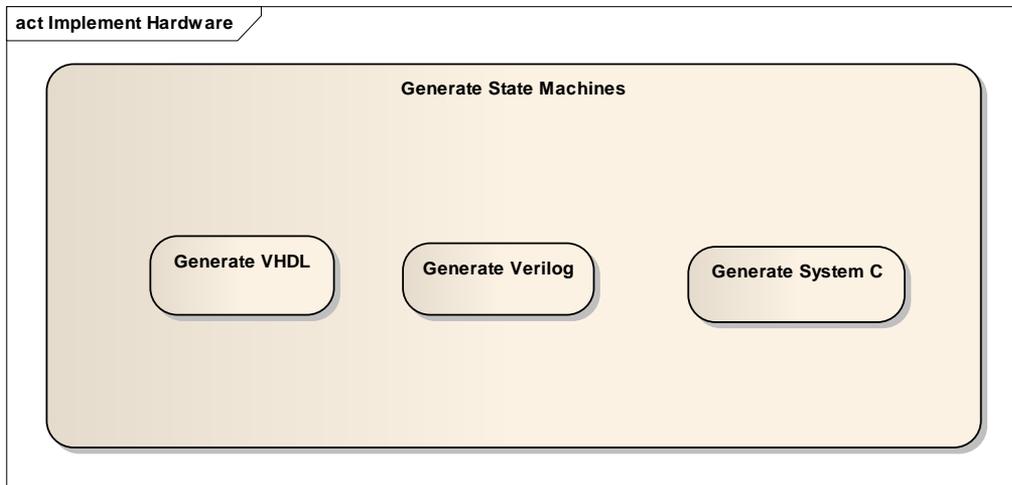
## Chapter 6

# Audio Player Hardware Implementation

We discussed Behavioral Modeling with State Machines in Chapter 3. In this Chapter, we'll demonstrate how to generate Hardware Description Language (HDL) code for State Machines, using our Audio Player Example. Then in Chapter 7 we'll explore software implementation.

### ***Hardware Implementation Roadmap***

Our Roadmap for implementing hardware via generating HDL code provides three parallel paths: implementation via VHDL, Verilog, and System-C. In all three cases, you'll leverage Enterprise Architect's unique ability to generate code from State Machines, and its powerful code-generation template capability.

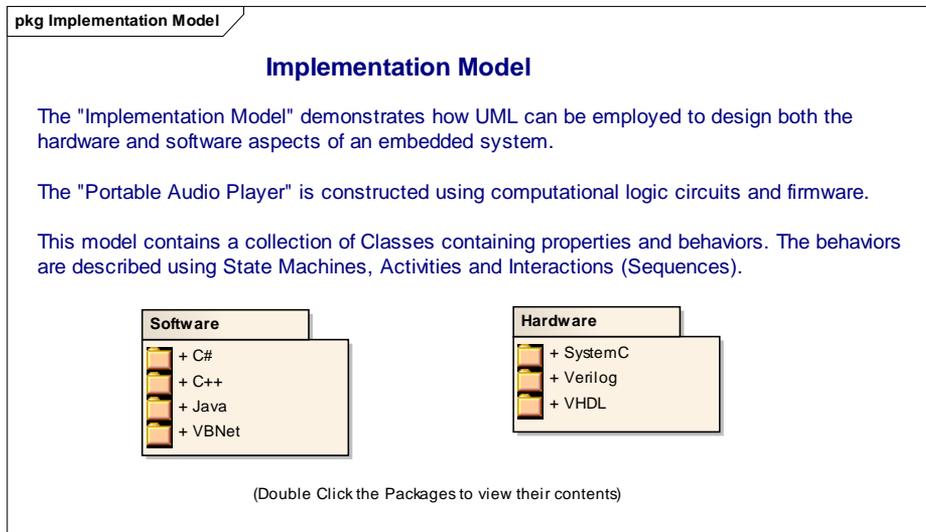


**Figure 1 – Hardware Implementation Roadmap with support for three popular Hardware Description Languages**

### ***Audio Player Hardware Implementation***

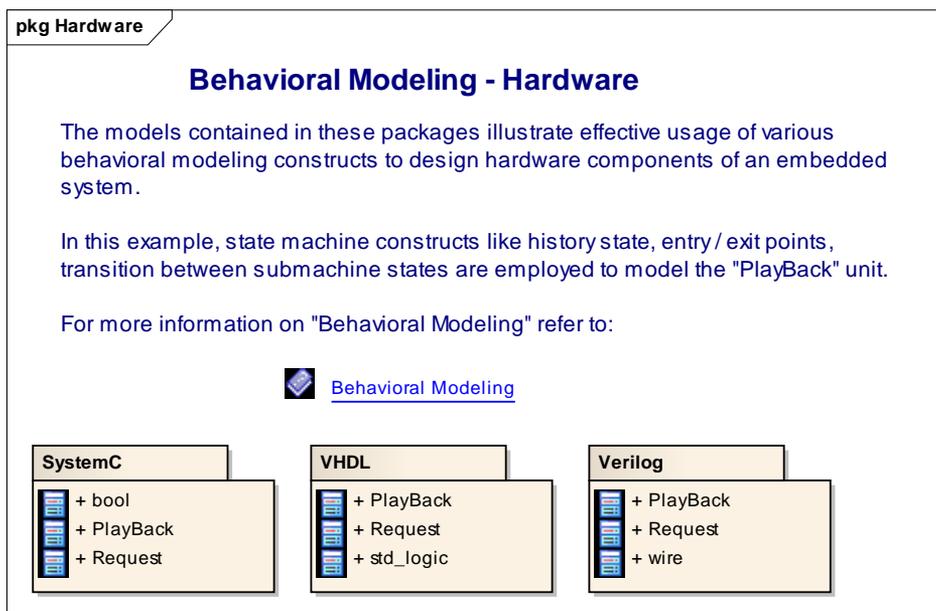
As usual, we'll illustrate our Roadmap using the Audio Player example. In this case, we'll explore the "Playback" operation and illustrate its implementation in VHDL, Verilog, and SystemC.

Figure 2 shows the top level package organization of the "Implementation" part of our Audio Player Model. We'll explore the software package in the next chapter. For the remainder of this chapter, we'll discuss code generation for State Machines, and present three flavors of generated HDL code for Playback.



**Figure 2 – Top Level Implementation Package**

Let's explore the Hardware package in more detail.



**Figure 3 – Enterprise Architect can generate HDL code for several languages.**

As you'll see, all three of our State Machine implementations use a common design pattern. In each case, the Playback class contains a state machine with On and Off states. The On state contains a child diagram (sub-state-machine) that contains the actual design.

There are three steps in building an HDL State Machine model:

1. Designate Driving Triggers
2. Establish Port–Trigger Mapping
3. Define Active State Logic

Let's look at each of these in turn.

# 1. Designate Driving Triggers

The top level State Machine diagram should be used to model the different modes of a hardware component, and the associated triggers that drive them, as shown in Figure 4.

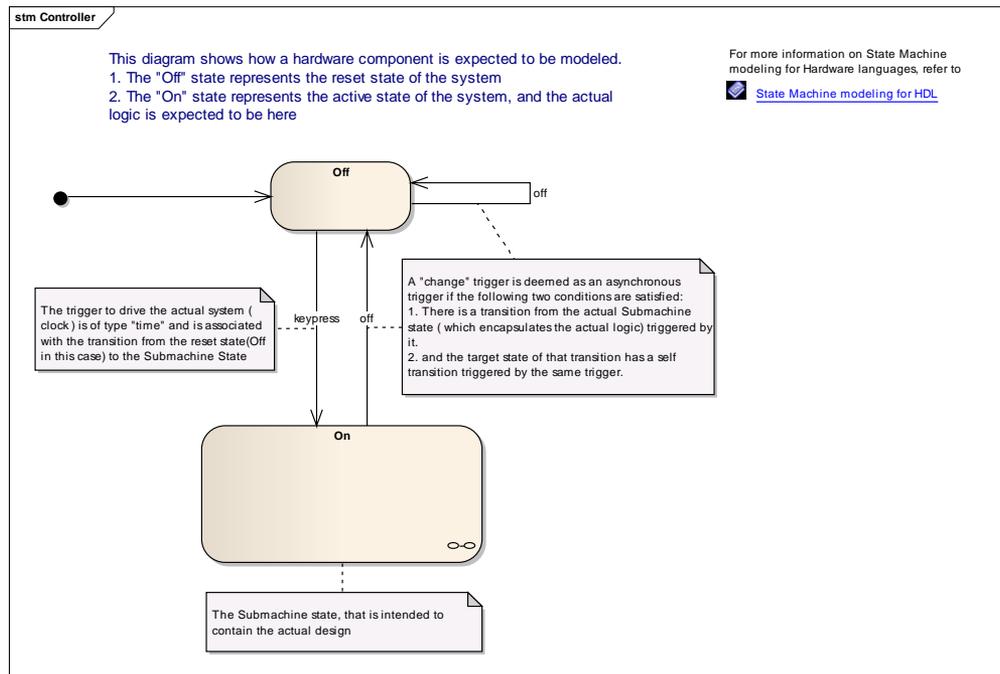


Figure 4 – The top level state machine is used to designate operating modes and driving triggers

There are several type of triggers.

## Asynchronous Triggers

Asynchronous triggers should be modeled according to the following pattern:

- The trigger should be of type *Change* (specification: **true / false**).
- The active state (Submachine State) should have a transition trigger by it.
- The target state of the triggered transition should have a self transition with the same trigger.

## Clock

A trigger of type *time*, which triggers the transitions to the active state (Submachine State) is deemed as the *Clock*. The specification of this trigger should be specific to the target language.

| Trigger Type | Language | Specification           |                         |
|--------------|----------|-------------------------|-------------------------|
|              |          | Positive Edge Triggered | Negative Edge Triggered |
| Time         | VHDL     | rising_edge             | falling_edge            |
|              | Verilog  | posedge                 | negedge                 |
|              | SystemC  | positive                | negative                |

Figure 5 - Clock Trigger Specifications

## 2. Establish Port – Trigger Mapping

After successfully modeling the different operating modes of the component, and the triggers associated with them, you must associate the triggers with the component's ports as shown in Figure 6.

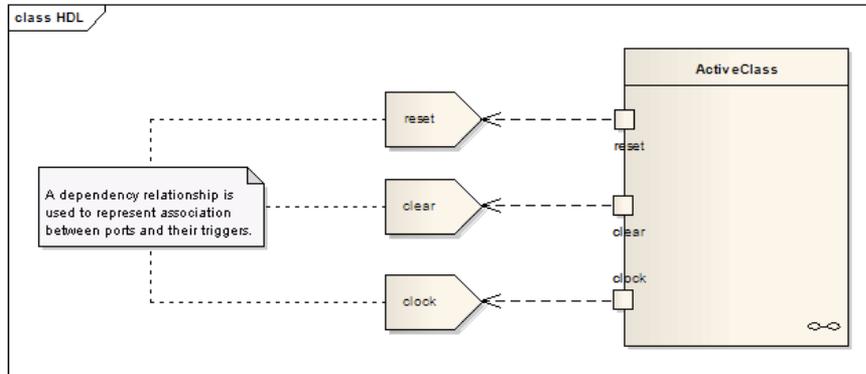


Figure 6 – Dependency relationships are used to map ports to triggers

## 3. Define Active State Logic

The first two aspects, above, put in place the preliminaries required for efficient interpretation of the hardware components. The actual State Machine logic is now modeled within the *Active* (Submachine) state.

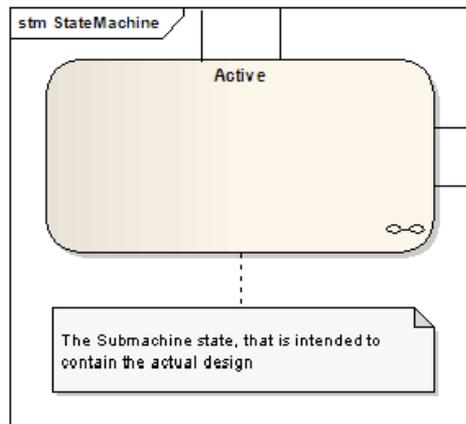


Figure 7 – Active logic is specified on the child submachine for the Active state

We'll explore Step 3 in some detail for VHDL, Verilog, and SystemC.

## Implementation in VHDL

Figure 8 shows a class diagram for Playback, with input and output ports designated. The design of the Playback functionality is contained in a multi-level state machine that's nested within the Playback class.

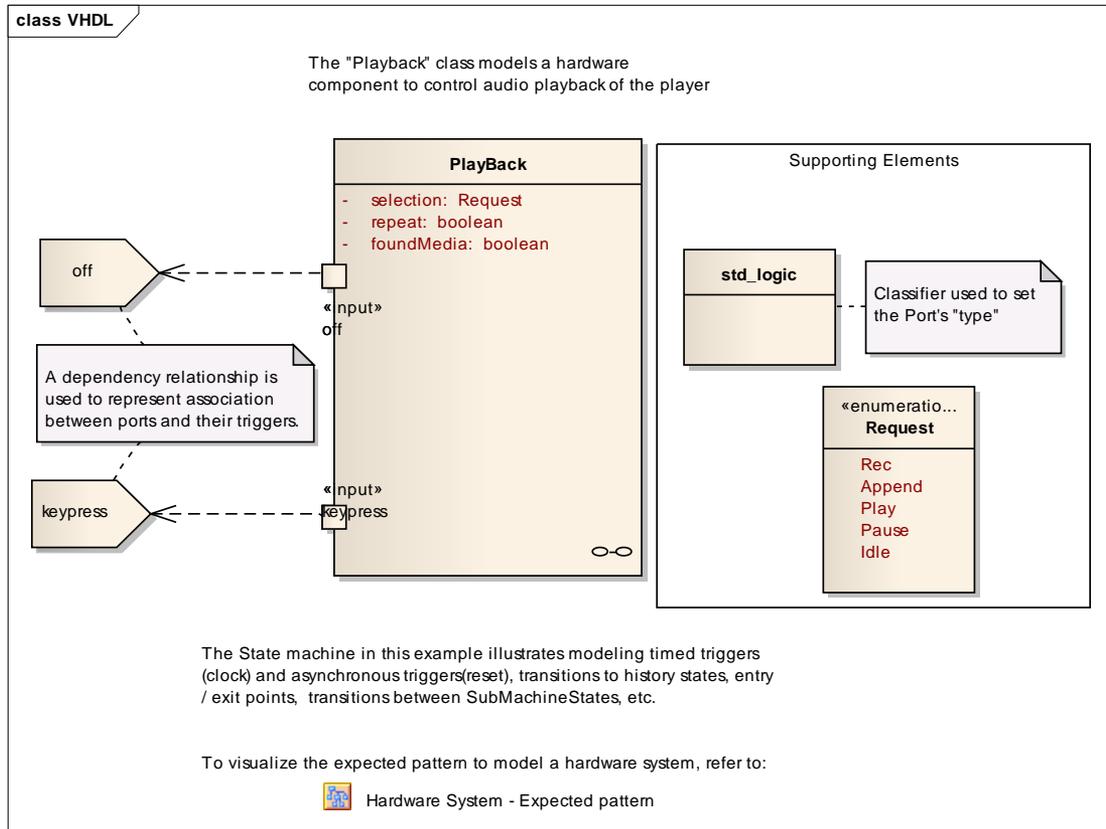
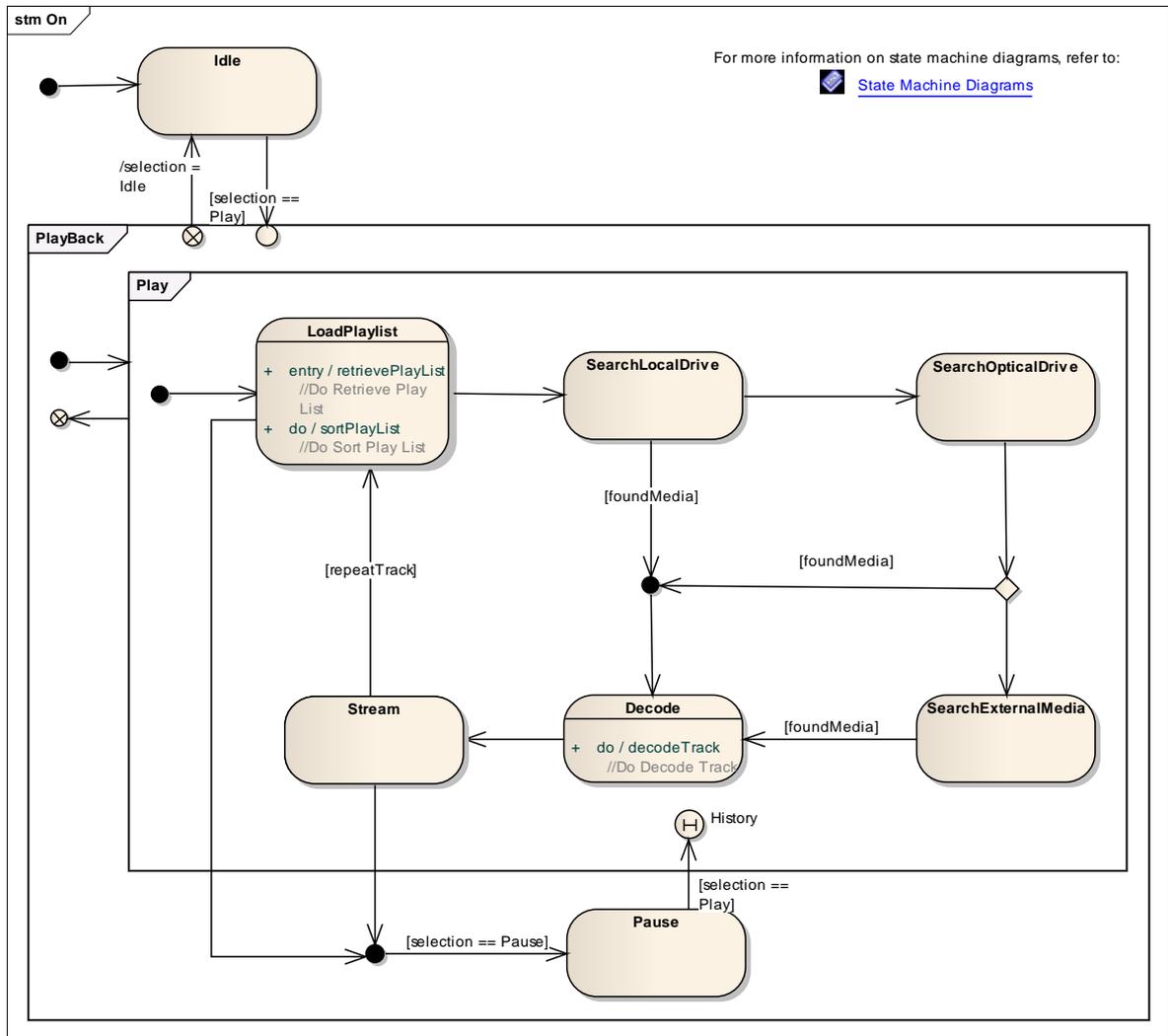


Figure 8 – VHDL code will be generated from substates nested within the Playback class



**Figure 9 – State Machine for Playback**

The State Machine shown in Figure 9 is essentially the same for the Verilog and SystemC implementations. The differences are so minor that we won't repeat the diagram in the upcoming sections on those HDLs.

## VHDL Code Generation and Reverse Engineering

### An Overview of VHDL

*Here's a brief summary of VHDL that we extracted from Wikipedia.<sup>6</sup>*

VHDL (VHSIC (Very High Speed Integrated Circuits) hardware description language) is commonly used as a design-entry language for field-programmable gate arrays and application-specific integrated circuits in electronic design automation of digital circuits.

VHDL was originally developed at the behest of the US Department of Defense in order to document the behavior of the ASICs that supplier companies were including in equipment. That is to say, VHDL was developed as an alternative to huge, complex manuals which were subject to implementation-specific details.

The idea of being able to simulate this documentation was so obviously attractive that logic simulators were developed that could read the VHDL files. The next step was the development of logic synthesis tools that read the VHDL, and output a definition of the physical implementation of the circuit. Modern synthesis tools can extract RAM, counter, and arithmetic blocks out of the code, and implement them according to what the user specifies. Thus, the same VHDL code could be synthesized differently for lowest area, lowest power consumption, highest clock speed, or other requirements.

VHDL is a fairly general-purpose language, and it doesn't require a simulator on which to run the code. There are many VHDL compilers, which build executable binaries. It can read and write files on the host computer, so a VHDL program can be written that generates another VHDL program to be incorporated in the design being developed. Because of this general-purpose nature, it is possible to use VHDL to write a testbench that verifies the functionality of the design using files on the host computer to define stimuli, interacts with the user, and compares results with those expected.

The key advantage of VHDL when used for systems design is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).

Another benefit is that VHDL allows the description of a concurrent system (many parts, each with its own sub-behavior, working together at the same time). VHDL is a Dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

A final point is that when a VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device such as a CPLD or FPGA, then it is the actual hardware being configured, rather than the VHDL code being "executed" as if on some form of a processor chip.

---

<sup>6</sup> <http://en.wikipedia.org/wiki/VHDL>

Enterprise Architect supports round-trip engineering of VHDL, using the following Stereotypes and Tagged Values.

## Stereotypes

| Stereotype           | Applies To  | Corresponds To                             |
|----------------------|-------------|--|
| <b>architecture</b>  | Class       | An architecture                            |
| <b>asynchronous</b>  | Method      | An asynchronous process                    |
| <b>configuration</b> | Method      | A configuration                            |
| <b>enumeration</b>   | Inner Class | An <i>enum</i> type                        |
| <b>entity</b>        | Interface   | An entity                                  |
| <b>part</b>          | Attribute   | A component instantiation                  |
| <b>port</b>          | Attribute   | A port                                     |
| <b>signal</b>        | Attribute   | A signal declaration                       |
| <b>struct</b>        | Inner Class | A record definition                        |
| <b>synchronous</b>   | Method      | A synchronous process                      |
| <b>typedef</b>       | Inner Class | A <i>type</i> or <i>subtype</i> definition |

Figure 10 – VHDL Stereotypes used by Enterprise Architect

## Tagged Values

| Tag                  | Applies To            | Corresponds To   |
|----------------------|-----------------------|--|
| <b>isGeneric</b>     | Attribute (port)      | The port declaration in a generic interface  |
| <b>isSubType</b>     | Inner Class (typedef) | A subtype definition   |
| <b>kind</b>          | Attribute (signal)    | The signal kind (e.g. <i>register</i> , <i>bus</i> )                                     |
| <b>mode</b>          | Attribute (port)      | The port mode ( <i>in</i> , <i>out</i> , <i>inout</i> , <i>buffer</i> , <i>linkage</i> ) |
| <b>portmap</b>       | Attribute (part)      | The generic / port map of the component instantiated                                     |
| <b>sensitivity</b>   | Method (synchronous)  | The sensitivity list of a synchronous process  |
| <b>type</b>          | Inner Class (typedef) | The type indication of a type declaration  |
| <b>typeNameSpace</b> | Attribute (part)      | The type namespace of the instantiated component   |

Figure 11 – VHDL Tagged Values used by Enterprise Architect

Figures 12 and 13 show a portion of the generated code produced by Enterprise Architect.

```

1  -----
2  -- Playback.vhdl
3  -- Implementation of the Class Playback
4  -- Created on:    11-Aug-2009 6:29:06 PM
5  -- Original author: nmizam
6  -----
7  library IEEE;
8  use IEEE.std_logic_1164.all;
9  use IEEE.std_logic_arith.all;
10 use IEEE.std_logic_textio.all;
11 use STD.textio.all;
12 entity Playback is
13 Port
14 (
15     off : in std_logic;
16     keypress : in std_logic
17 );
18 end Playback;
19 architecture Controller of Playback is
20 signal selection : Request;
21 signal foundMedia : boolean;
22 signal repeat : boolean;
23
24 type StateType is
25 (
26     Controller_Off,
27     Controller_On,
28     Controller_On_PlayBack,
29     Controller_On_PlayBack_Pause,
30     Controller_On_PlayBack_Play,
31     Controller_On_PlayBack_Play_Decode

```

**Figure 12 – Enterprise Architect generates VHDL code from a state machine**

Note that the VHDL code generated is extremely detailed and robust.

```

74     end process CLOCK_DIV2;
75
76     CLOCK_DIV4 : process(clockDiv2 , off)
77     begin
78         if(off = '1') then
79             clockDiv4 <= '1';
80         elsif(rising_edge(clockDiv2)) then
81             clockDiv4 <= not clockDiv4;
82         end if;
83     end process CLOCK_DIV4;
84
85     TRANSITION_LOGIC: process(currState , off)
86     variable bFlag : boolean; begin
87         currTransition <= TT_NOTTRANSITION;
88         if(off = '1') then
89             nextState <= Controller_Off;
90             currTransition <= TT_NOTTRANSITION;
91         else
92             case currState is
93             when Controller_Off =>
94                 bFlag := false;
95                 nextState <= Controller_On_Idle;
96                 transcend <= '1';
97                 Controller_history <= currState;
98             when Controller_On =>
99                 nextState <= Controller_On_Idle;
100                currTransition <= Controller_On_PlayBack_to_Controller_On_Idle;
101                transcend <= '1';
102                Controller_On_PlayBack_history <= currState;
103             when Controller_On_PlayBack =>
104                 nextState <= Controller_On_Idle;

```

**Figure 13 – Enterprise Architect’s state-machine code generator, combined with SysML parts and ports, and VHDL stereotypes and tagged values, produces a very complete implementation.**

## Implementation in Verilog

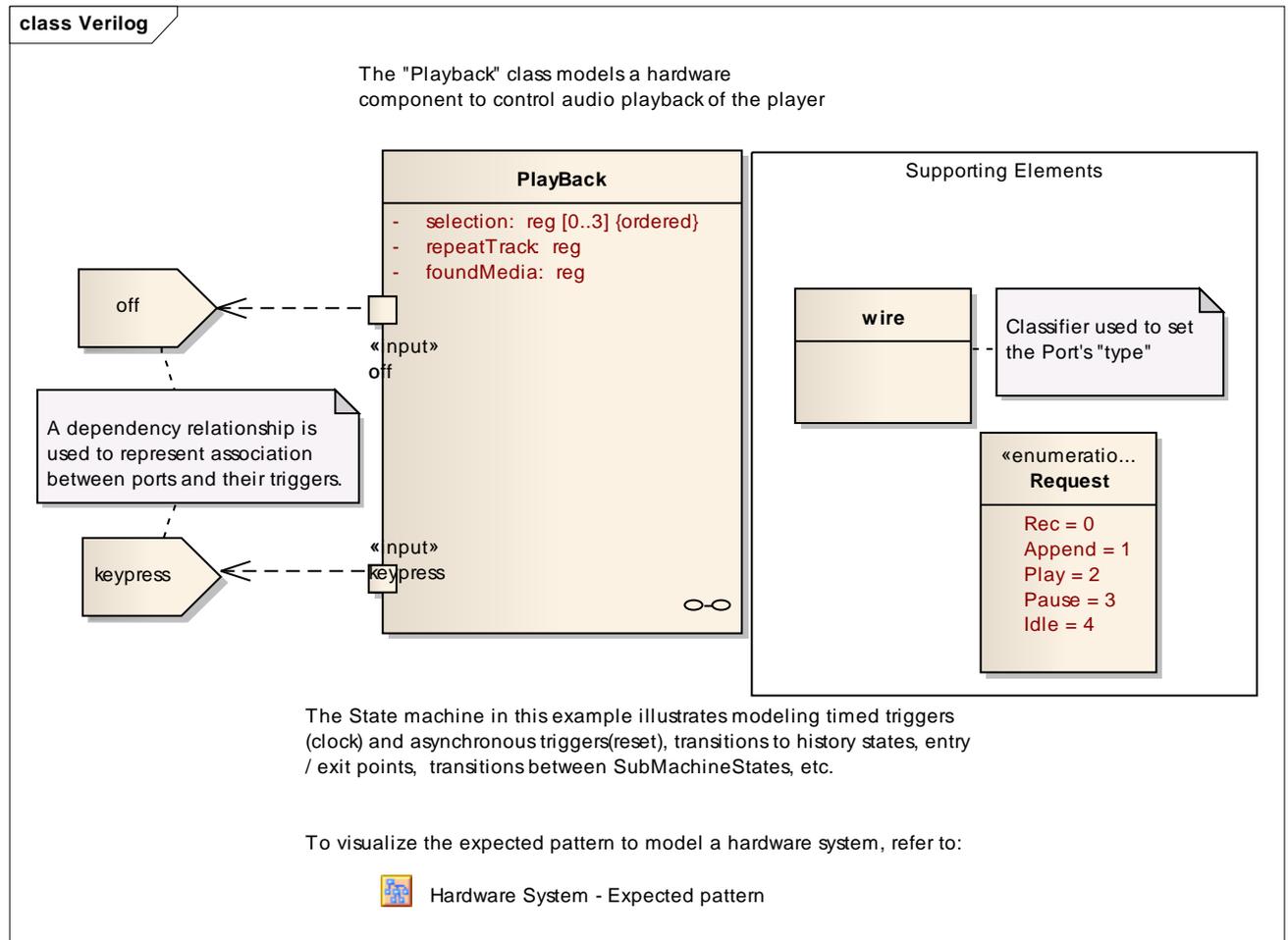


Figure 14 – Playback class diagram for Verilog implementation

## An Overview of Verilog

We consulted Wikipedia again<sup>7</sup> for an overview of Verilog.

In the semiconductor and electronic design industry, Verilog is a hardware description language (HDL) used to model electronic systems. Verilog HDL, not to be confused with VHDL, is most commonly used in the design, verification, and implementation of digital logic chips at the Register transfer level (RTL) level of abstraction. It is also used in the verification of analog and mixed-signal circuits.

Hardware description languages, such as Verilog, differ from software programming languages because they include ways of describing the propagation of time and signal dependencies (sensitivity). There are two assignment operators, a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of the Verilog's language semantics, designers could

<sup>7</sup> <http://en.wikipedia.org/wiki/Verilog>

quickly write descriptions of large circuits, in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic-capture, and specially-written software programs to document and simulate electronic circuits.

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But the blocks themselves are executed concurrently, qualifying Verilog as a Dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined"), and strengths (strong, weak, etc.) This system allows abstract modeling of shared signal-lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements in the Verilog language is synthesizable. Verilog modules that conform to a synthesizable coding-style, known as RTL (register transfer level), can be physically realized by synthesis software. Synthesis-software algorithmically transforms the (abstract) Verilog source into a netlist, a logically-equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flipflops, etc.) that are available in a specific VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask-set for an ASIC, or a bitstream-file for an FPGA).

Enterprise Architect supports round-trip engineering of Verilog code, using the following Stereotypes and Tagged Values.

## Stereotypes

| Stereotype          | Applies To  | Corresponds To            |
|---------------------|-------------|---------------------------|
| <b>asynchronous</b> | Method      | A concurrent process      |
| <b>enumeration</b>  | Inner Class | An <i>enum</i> type       |
| <b>initializer</b>  | Method      | An initializer process    |
| <b>module</b>       | Class       | A module                  |
| <b>part</b>         | Attribute   | A component instantiation |
| <b>port</b>         | Attribute   | A port                    |
| <b>synchronous</b>  | Method      | A sequential process      |

Figure 15 – Verilog Stereotypes used by Enterprise Architect

## Tagged Values

| Tag         | Applies To         | Corresponds To  |
|-------------|--------------------|---|
| kind        | Attribute (signal) | The signal kind (such as <i>register</i> , <i>bus</i> ) |
| mode        | Attribute (port)   | The port mode ( <i>in</i> , <i>out</i> , <i>inout</i> ) |
| Portmap     | Attribute (part)   | The generic / port map of the component instantiated    |
| sensitivity | Method             | The sensitivity list of a sequential process            |
| type        | Attribute          | The range or type value of an attribute                 |

Figure 16 – Verilog Tagged Values used by Enterprise Architect

```

1 ////////////////////////////////////////////////////////////////////
2 // Playback.v
3 // Implementation of the Class Playback
4 // Created on:      11-Aug-2009 6:32:44 PM
5 // Original author: mnizam
6 ////////////////////////////////////////////////////////////////////
7
8
9 module Playback
10 {
11     keypress,
12     off
13 };
14 parameter SIZE = 4;
15 input wire keypress;
16 input wire off;
17
18 parameter /*(StateType)*/
19     Controller_Off = 0,
20     Controller_On = 1,
21     Controller_On_PlayBack = 2,
22     Controller_On_PlayBack_Pause = 3,
23     Controller_On_PlayBack_Play = 4,
24     Controller_On_PlayBack_Play_SearchLocalDrive = 5,
25     Controller_On_PlayBack_Play_LoadPlaylist = 6,
26     Controller_On_PlayBack_Play_Decode = 7,
27     Controller_On_PlayBack_Play_SearchExternalMedia = 8,
28     Controller_On_PlayBack_Play_SearchOpticalDrive = 9,
29     Controller_On_PlayBack_Play_Stream = 10,
30     Controller_On_Idle = 11,
31     STATE = 12;

```

Figure 17 – Verilog code generated by Enterprise Architect

## Implementation in SystemC

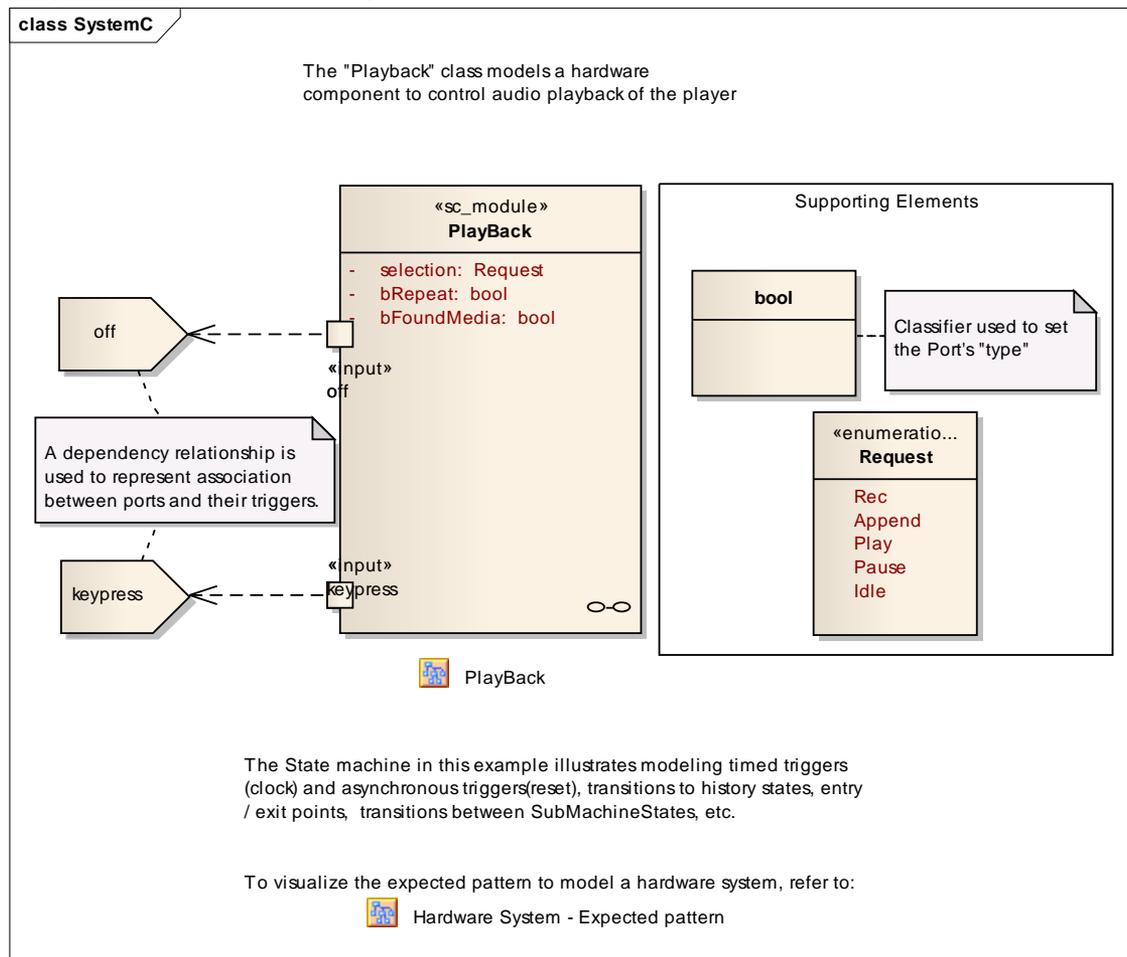


Figure 18 – Playback class diagram for SystemC implementation

## An Overview of SystemC

We've consulted Wikipedia one final time<sup>8</sup> for our overview of SystemC.

SystemC is a set of C++ classes and macros which provide an event-driven simulation kernel in C++ (see also discrete event simulation). SystemC makes it possible to simulate concurrent processes, each described using plain C++ syntax. SystemC processes can communicate in a simulated real-time environment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as user defined. In certain respects, SystemC deliberately mimics the hardware description languages VHDL and Verilog, but is more aptly described as a system- level modeling language.

SystemC is used for system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. SystemC is often associated with Electronic system level (ESL) design, and with Transaction-level modeling (TLM).

<sup>8</sup> [http://en.wikipedia.org/wiki/System\\_C](http://en.wikipedia.org/wiki/System_C)

SystemC is defined and promoted by OSCI, the Open SystemC Initiative. OSCI also provide an open-source proof-of-concept simulator (sometimes incorrectly referred to as the reference simulator), which can be downloaded from the OSCI website<sup>9</sup>.

SystemC has semantic similarities to VHDL and Verilog, but may be said to have a syntactical overhead compared to these when used as a hardware description language. On the other hand, greater freedom of expressiveness is offered in return, like object oriented design partitioning and template classes. Although strictly a C++ class library, SystemC is sometimes viewed as being a language in its own right. Source code can be compiled with the SystemC library (which includes a simulation kernel) to give an executable. The performance of the OSCI open-source implementation is typically less optimal than commercial VHDL/Verilog simulators when used for register transfer level simulation.

SystemC version 1 included common hardware description language features such as structural hierarchy and connectivity, clock cycle accuracy, delta cycles, 4-state logic (0, 1, X, Z), and bus resolution functions. From version 2 onward, the focus of SystemC has moved to communication abstraction, transaction-level modeling, and virtual platform modeling. SystemC version 2 added abstract ports, dynamic processes, and timed event notifications.

Enterprise Architect supports round-trip engineering of SystemC code, using the following Stereotypes and Tagged Values.

## Stereotypes

| Stereotype         | Applies To  | Corresponds To                    |
|--------------------|-------------|-----------------------------------|
| <b>delegate</b>    | Method      | A delegate.                       |
| <b>enumeration</b> | Inner Class | An <i>enum</i> type.              |
| <b>friend</b>      | Method      | A <i>friend</i> method.           |
| <b>property</b>    | Method      | A property definition.            |
| <b>sc_ctor</b>     | Method      | A SystemC constructor.            |
| <b>sc_module</b>   | Class       | A SystemC module.                 |
| <b>sc_port</b>     | Attribute   | A port.                           |
| <b>sc_signal</b>   | Attribute   | A signal                          |
| <b>struct</b>      | Inner Class | A <i>struct</i> or <i>union</i> . |

**Figure 19 – SystemC stereotypes used by Enterprise Architect**

<sup>9</sup> <http://www.systemc.org/home/>

## Tagged Values

| Tag       | Applies To       | Corresponds To   |
|-----------|------------------|--|
| kind      | Attribute (Port) | Port kind ( <i>clocked, fifo, master, slave, resolved, vector</i> ). |
| mode      | Attribute (Port) | Port mode ( <i>in, out, inout</i> ).                                 |
| overrides | Method           | The <i>Inheritance</i> list of a method declaration.                 |
| throw     | Method           | The exception specification of a method.                             |

Figure 20 – SystemC tagged values used by Enterprise Architect

```

1 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2 // Playback.sc
3 // Implementation of the Class Playback
4 // Created on:      11-Aug-2009 6:31:54 PM
5 // Original author: mnizam
6 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
7
8 #include "systemc.h"
9 SC_MODULE ( Playback )
10 {
11     Request selection;
12     bool   bRepeat;
13     bool   bFoundMedia;
14
15     sc_in_clk  keypress;
16     sc_in <bool> off;
17     /* -----Begin - Code rendered by EA for the underlying Behavioral Model-----
18
19     enum StateType
20     {
21         Controller_On,
22         Controller_On_PlayBack,
23         Controller_On_PlayBack_Pause,
24         Controller_On_PlayBack_Play,
25         Controller_On_PlayBack_Play_SearchLocalDrive,
26         Controller_On_PlayBack_Play_SearchExternalMedia,
27         Controller_On_PlayBack_Play_SearchOpticalDrive,
28         Controller_On_PlayBack_Play_Decompile,
29         Controller_On_PlayBack_Play_LoadPlaylist,
30         Controller_On_PlayBack_Play_Stream,
31         Controller_On_Idle

```

Figure 21 – SystemC code generated by Enterprise Architect

## Chapter 7

# Audio Player Software Implementation

Enterprise Architect contains numerous features to help with code generation and reverse engineering, and also integrates closely with the Visual Studio and Eclipse development environments via its MDG Integration technology. Many of Enterprise Architect's code engineering capabilities, including forward and reverse engineering, and Enterprise Architect's powerful code template framework, are described in detail in the [Enterprise Architect for Power Users](#) multimedia tutorial.<sup>10</sup> This chapter will focus in on Sparx Systems' unique capability for *Behavioral Code Generation*, and on the MDG Integration capability. Figure 1 shows our Roadmap for Software Implementation.

## Software Implementation Roadmap

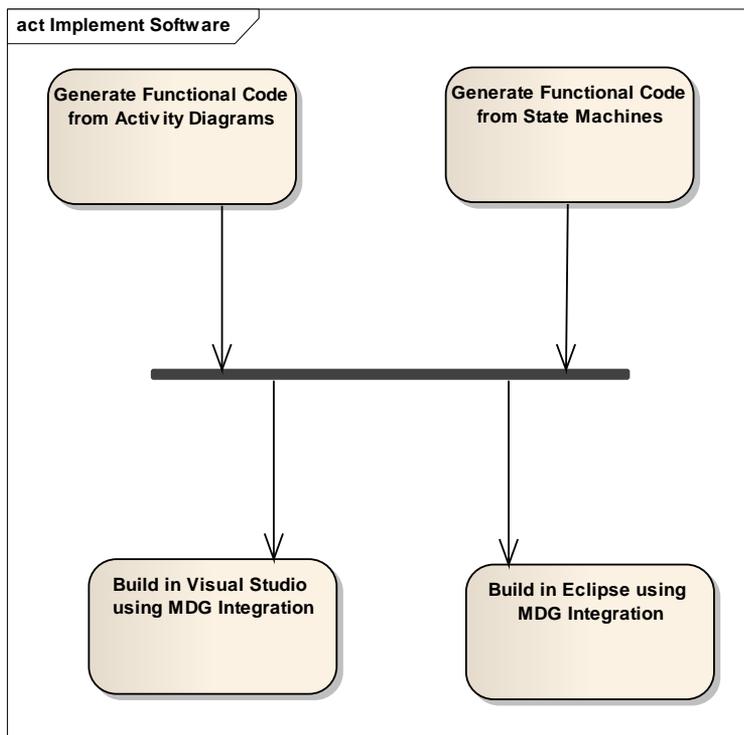


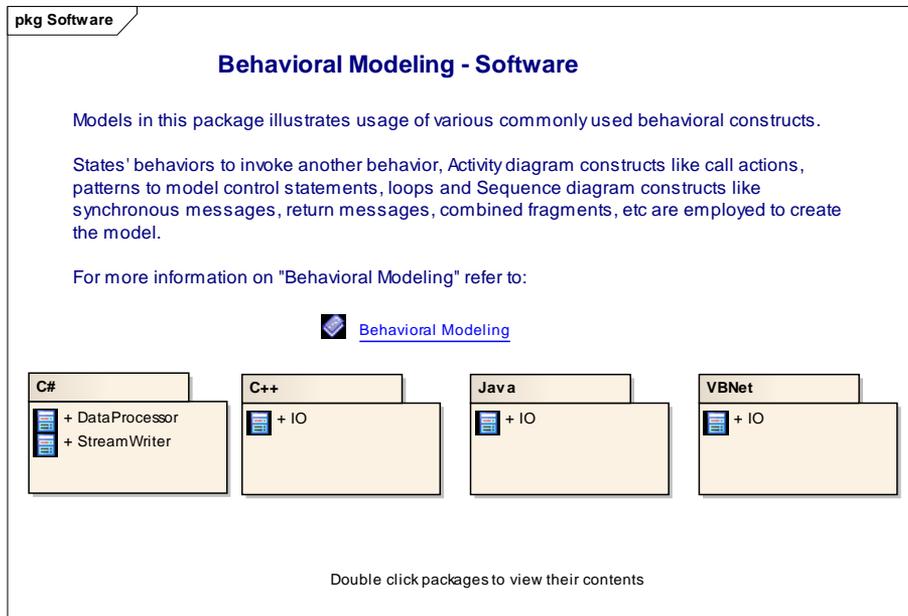
Figure 1 – Roadmap for Software Implementation

## Behavioral Models can be code generated

Enterprise Architect enables you to define an element's behavior through the element's operations and parameters. You can also define the behavior of more specific behavioral elements such as Activities, Interactions, Actions and Interaction Occurrences.

In this chapter, we'll explore how to transform behavior models of the type that you saw in Chapter 3 into executable source code in C#, C++, Java, and Visual Basic. Figure 2 shows the top level package diagram from our audio player example, which we'll use to illustrate behavioral code generation.

<sup>10</sup> *Enterprise Architect for Power Users* multimedia tutorial: [www.iconixsw.com/EA/PowerUsers.html](http://www.iconixsw.com/EA/PowerUsers.html)



**Figure 2 – Audio Player example organization for behavioral code generation**

In addition to its long-standing ability to generate code for software classes, Enterprise Architect supports generation of code from three UML behavioral modeling behavioral paradigms:

- *State Machine diagrams*
- *Interaction diagrams*
- *Activity diagrams*

We'll explore behavioral code generation in considerable detail in this chapter, and it should be an interesting ride to some places you've probably never been to before, so fasten your seat belts. We'll start off with a look at generating C# code from state machines and activity diagrams, for the `DataProcessor` class from our audio player.

Figure 3 shows a class diagram for the `DataProcessor` class, which contains a nested state machine for `Searching External Media`, and a nested Activity Diagram for `Appending to a Buffer`. Figure 4 shows the nesting of behaviors on a Composite Structure Diagram.

# Data Processor: C# code gen from State and Activity Diagrams

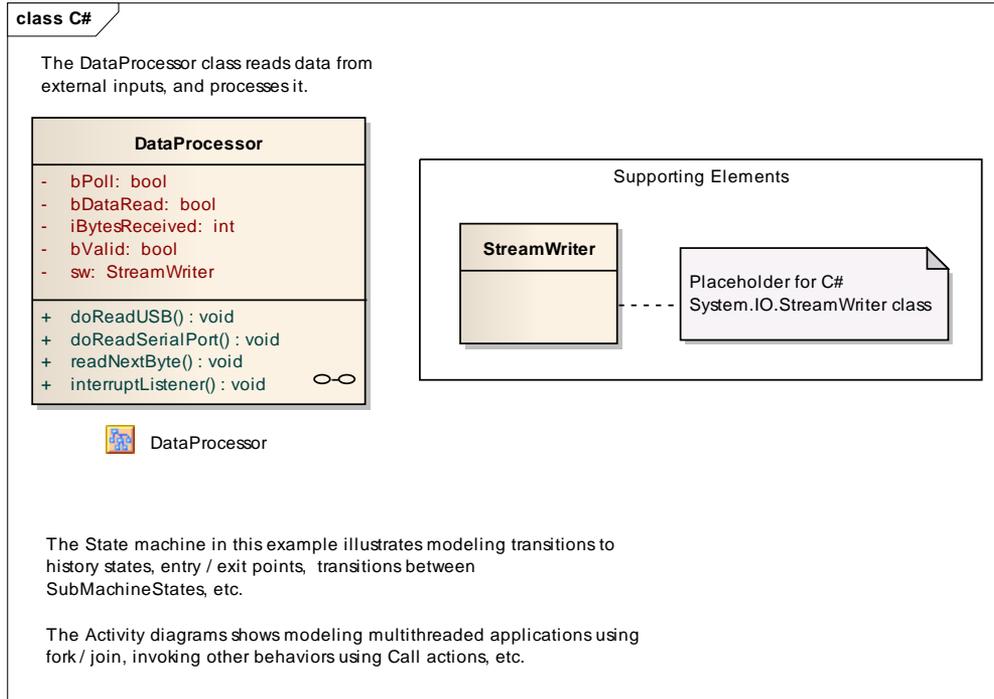


Figure 3 – DataProcessor Class Diagram

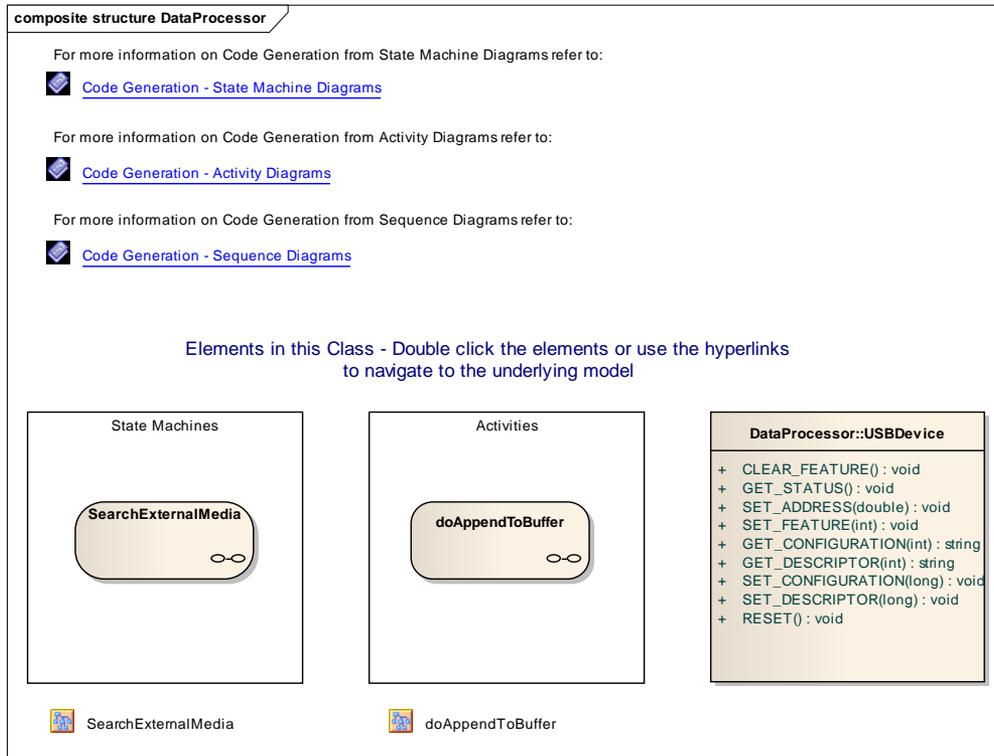


Figure 4 – Composite Structure Diagram illustrating the nested behaviors of DataProcessor

Behavioral code generation in Enterprise Architect requires the behavioral diagrams to be nested within the “Active Class” (the class that gets generated). Figure 5 shows the organization in Enterprise Architect’s project browser.

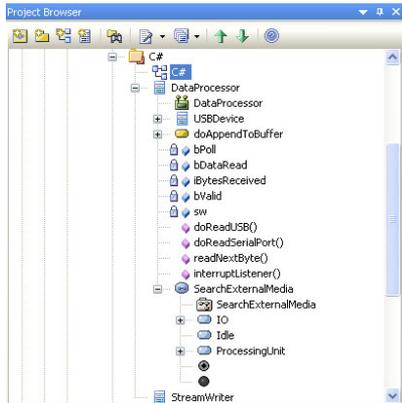


Figure 5 – Behaviors to be code generated are nested within a parent class

## Behavioral Code Generation from State Machines

A State Machine that’s nested within a Class generates the following constructs to enable effective execution of the States’ *do*, *entry* and *exit* behaviors and also to code the appropriate transition’s effect when necessary.

### Enumerations

**StateType** – comprises an enumeration for each of the States contained within the State Machine

**TransitionType** – comprises an enumeration for each transition that has a valid effect associated with it, e.g. *ProcessOrder\_Delivered\_to\_ProcessOrder\_Closed*

**CommandType** – comprises an enumeration for each of the behavior types that a State can contain (*Do*, *Entry*, *Exit*).

### Attributes

**currState:StateType** – a variable to hold the current State’s information

**nextState:StateType** – a variable to hold the next State’s information, set by each State’s transitions accordingly

**currTransition:TransitionType** – a variable to hold the current transition information; this is set if the transition has a valid effect associated with it

**transcend:Boolean** – a flag used to advise if a transition is involved in transcending between different State Machines (or Submachine states)

**xx\_history:StateType** – a history variable for each State Machine/Submachine State, to hold information about the last State from which the transition took place.

### Operations

**StatesProc** – a States procedure, containing a map between a State’s enumeration and its operation; it de-references the current State’s information to invoke the respective State’s function

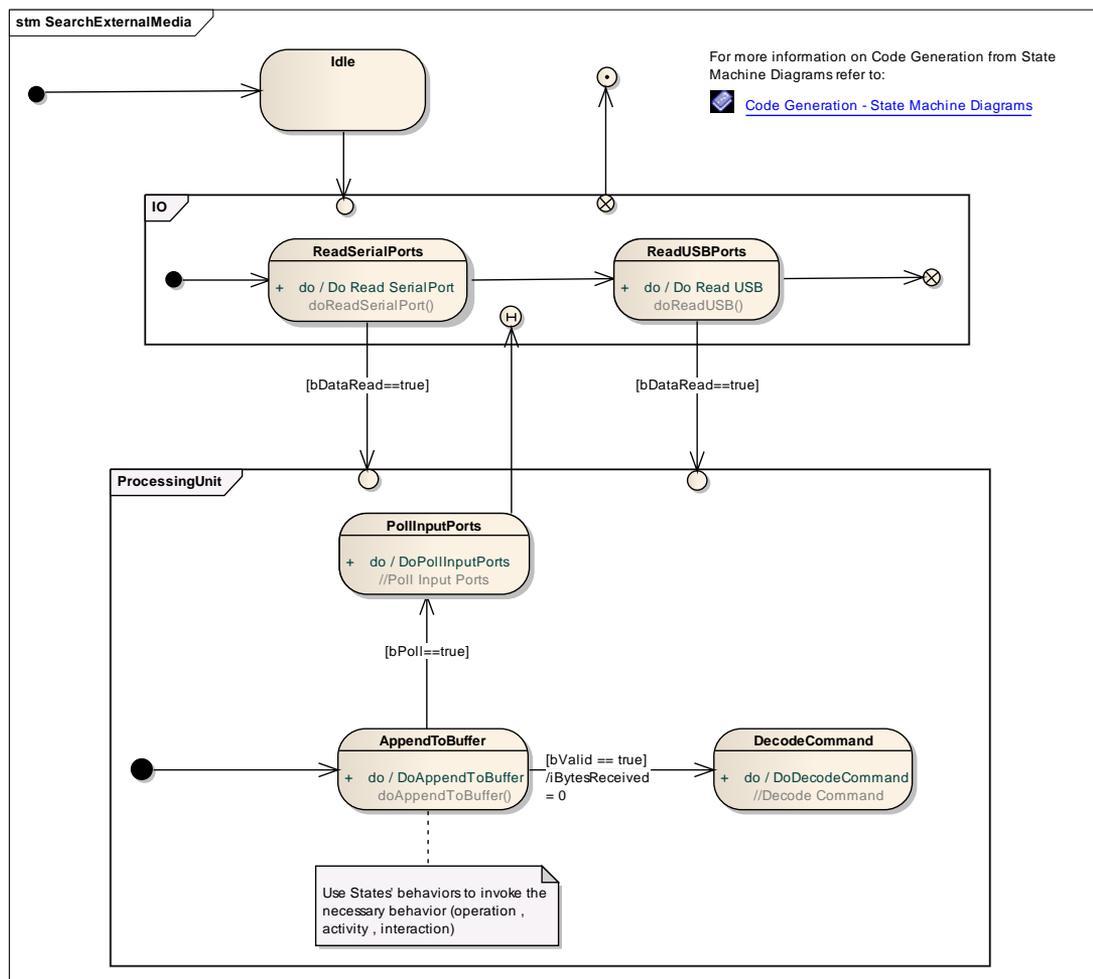
**TransitionsProc** – a Transitions procedure, containing a map between the Transition’s enumeration and its effect; it invokes the respective effect

<<State>> – an operation for each of the States contained within the State Machine; this renders a State's behaviors based on the input *CommandType*, and also executes its transitions

**initializeStateMachine** – a function that initializes all the framework-related attributes

**runStateMachine** – a function that iterates through each State, and executes their behaviors and transitions accordingly.

Figure 6 shows the state machine for *SearchExternalMedia*, and you can see a bit of the automatically generated code for the Do, Entry, and Exit states in Figure 7. The complete behavior of the state machine is generated automatically.



**Figure 6 – State Machine for Searching External Media**

The ProcessingUnit Polls its Input Ports, Appends to a Buffer, and Decodes Commands. We'll see the nested activity diagram and generated code for AppendToBuffer in Figures 8 and 9. But first, Figure 7 shows generated code for the state machine shown in Figure 6.

```
155      /* Begin - EA generated code for StateMachine */
156
157
158
159      private enum StateType : int
160      {
161          SearchExternalMedia,
162          SearchExternalMedia_ProcessingUnit,
163          SearchExternalMedia_ProcessingUnit_PollInputPorts,
164          SearchExternalMedia_ProcessingUnit_AppendToBuffer,
165          SearchExternalMedia_ProcessingUnit_DecodeCommand,
166          SearchExternalMedia_IO,
167          SearchExternalMedia_IO_ReadUSBPorts,
168          SearchExternalMedia_IO_ReadSerialPorts,
169          SearchExternalMedia_Idle,
170          ST_NOSTATE
171      }
172
173      private enum TransitionType : int
174      {
175          SearchExternalMedia_ProcessingUnit_AppendToBuffer_to_SearchExternalM
176          TT_NOTRANSITION
177      }
178
179      private enum CommandType : int
180      {
181          Do,
182          Entry,
183          Exit
184      }
185
186      private StateType currState;
```

Figure 7 – Enterprise Architect generates Behavioral Code for State Machines

## Behavioral Code Generation from Activity Diagrams

Enterprise Architect uses a **system engineering graph optimizer** to analyze an activity diagram and render it into various code-generatable constructs. The constructs are also transformed into one of the various action types (if appropriate), similar to Interaction diagram constructs.

### Conditional Statements

To model a conditional statement, you use Decision/Merge nodes. Alternatively, you can imply Decisions/Merges internally. The graph optimizer expects an associated Merge node for each Decision node, to facilitate efficient tracking of various branches and analysis of the code constructs within them.

### Invocation Actions (Call Operation Action, Call Behavior Action)

Call Actions are handled more efficiently. Each action has arguments relating to the parameters of the associated behavior (use the Synchronize button of the Arguments dialog to synchronize arguments and parameters).

### **Atomic Actions**

Atomic actions contain implementation-specific statements that are rendered 'in line' as a sequence of code statements in a procedure.

### Loops

Enterprise Architect's system engineering graph optimizer is also capable of analyzing and identifying loops. An identified loop is internally rendered as an *Action Loop*, which is translated by the EASL code generation macros to generate the required code.

Figure 8 shows an activity diagram for AppendToBuffer, and Figure 9 shows a snip of the resulting C# code.

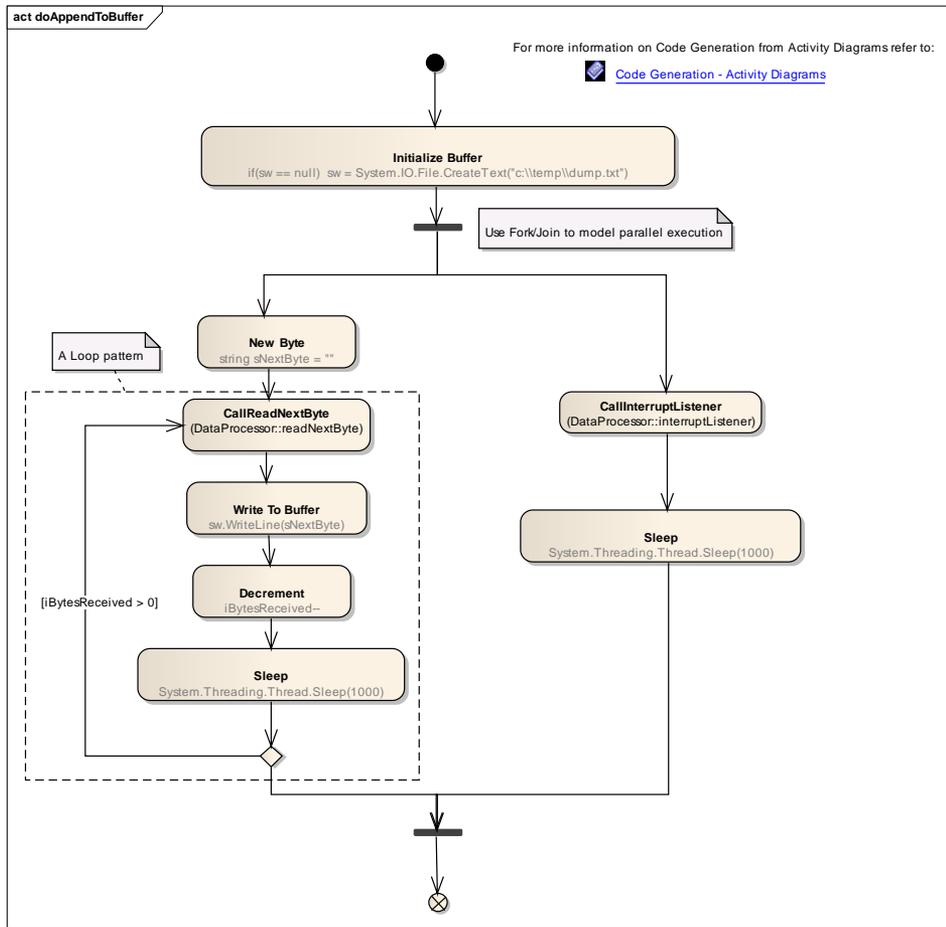


Figure 8 – Activity Diagram for AppendToBuffer

| File Explorer              | Line Number | Code |
|----------------------------|-------------|------|
| DataProcessor.cs           | 112         |      |
| USBDevice                  | 113         |      |
| «enumeration» CommandTy    | 114         |      |
| «enumeration» StateType    | 115         |      |
| «enumeration» TransitionTy | 116         |      |
| SearchExternalMedia_IO_H   | 117         |      |
| SearchExternalMedia_Proce  | 118         |      |
| SearchExternalMedia_histor | 119         |      |
| bDataRead                  | 120         |      |
| bPoll                      | 121         |      |
| bValid                     | 122         |      |
| currState                  | 123         |      |
| currTransition             | 124         |      |
| iBytesReceived             | 125         |      |
| nextState                  | 126         |      |
| sw                         | 127         |      |
| transcend                  | 128         |      |
| DataProcessor()            | 129         |      |
| Dispose()                  | 130         |      |
| StatesProc()               | 131         |      |
| TransitionsProc()          | 132         |      |
| doAppendToBuffer()         | 133         |      |
| doReadSerialPort()         | 134         |      |
| doReadUSB()                | 135         |      |
| initializeStateMachine()   | 136         |      |
| interruptListener()        | 137         |      |
| readNextByte()             | 138         |      |
| runStateMachine()          | 139         |      |
| searchExternalMedia_IO     | 140         |      |
| searchExternalMedia_IO_Rt  |             |      |
| searchExternalMedia_IO_Rt  |             |      |

```

112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
/* Begin - EA generated code for Activities and Interactions */
public void doAppendToBuffer()
{
    // behavior is a Activity
    if(sw == null) sw = System.IO.File.CreateText("c:\\temp\\dump.tx
System.Threading.Thread thread0 = new System.Threading.Thread(
new System.Threading.ThreadStart(
    delegate()
    {
        interruptListener();
        System.Threading.Thread.Sleep(1000);
    }
});
thread0.Start();

System.Threading.Thread thread1 = new System.Threading.Thread(
new System.Threading.ThreadStart(
    delegate()
    {
        string sNextByte = "";
        readNextByte();
        sw.WriteLine(sNextByte);
        iBytesReceived--;
        System.Threading.Thread.Sleep(1000);
        while (iBytesReceived > 0)

```

Figure 9 – Generated C# code for AppendToBuffer

Once again, the **full detail of the behavior detailed on the diagram is automatically generated into code.**

## IO – Code generation in C++, Java, and VB.Net

In this section, we'll look inside the IO class and explore behavioral code generation in C++, Java, and VB.Net. Figure 10 shows the class diagram for the C++ branch of the model; similar diagrams (not shown here) appear in the Java and VB packages.

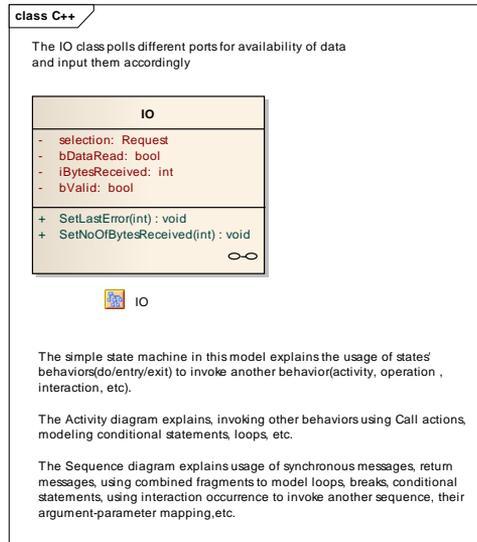


Figure 10 – IO class diagram

As with the DataProcessor example, all behaviors which we'd like to code generate are nested within the IO class.

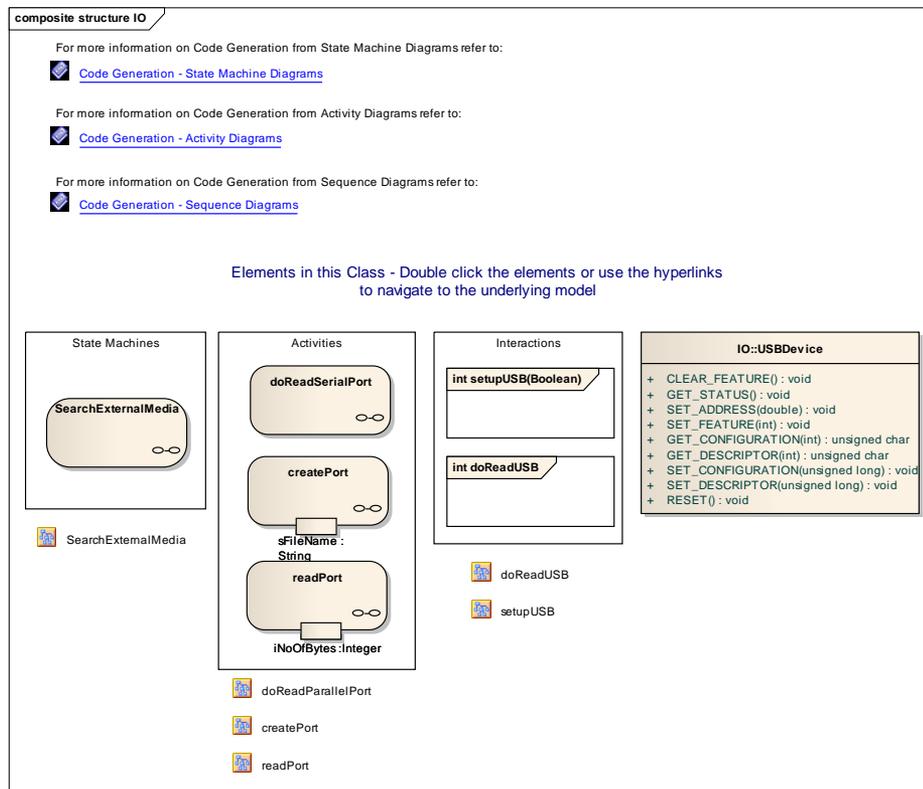


Figure 11 – Nested Behaviors of IO

In this section we'll explore code generation from Interaction, State, and Activity Diagrams. Figure 12 shows the sequence diagram for Setting up the USB port, and Figure 13 shows how to Read the USB port. Figure 14 shows a fragment of the automatically generated C++ code.

## Code Generation from Sequence Diagrams

Code generation from sequence diagrams that are nested within a Class uses Enterprise Architect's system engineering graph optimizer to transform the diagram into code.

**Messages** and **Fragments** are identified as one of the several action types based on their functionality, and the EASL code generation templates are used to render their behavior accordingly. For example:

A Message that invokes an operation is identified as an *Action Call* and is rendered accordingly

Combined Fragments are identified by their types and conditions; for instance, an *Alt* fragment is identified as an *Action If*, and a *loop* fragment is identified as an *Action Loop*.

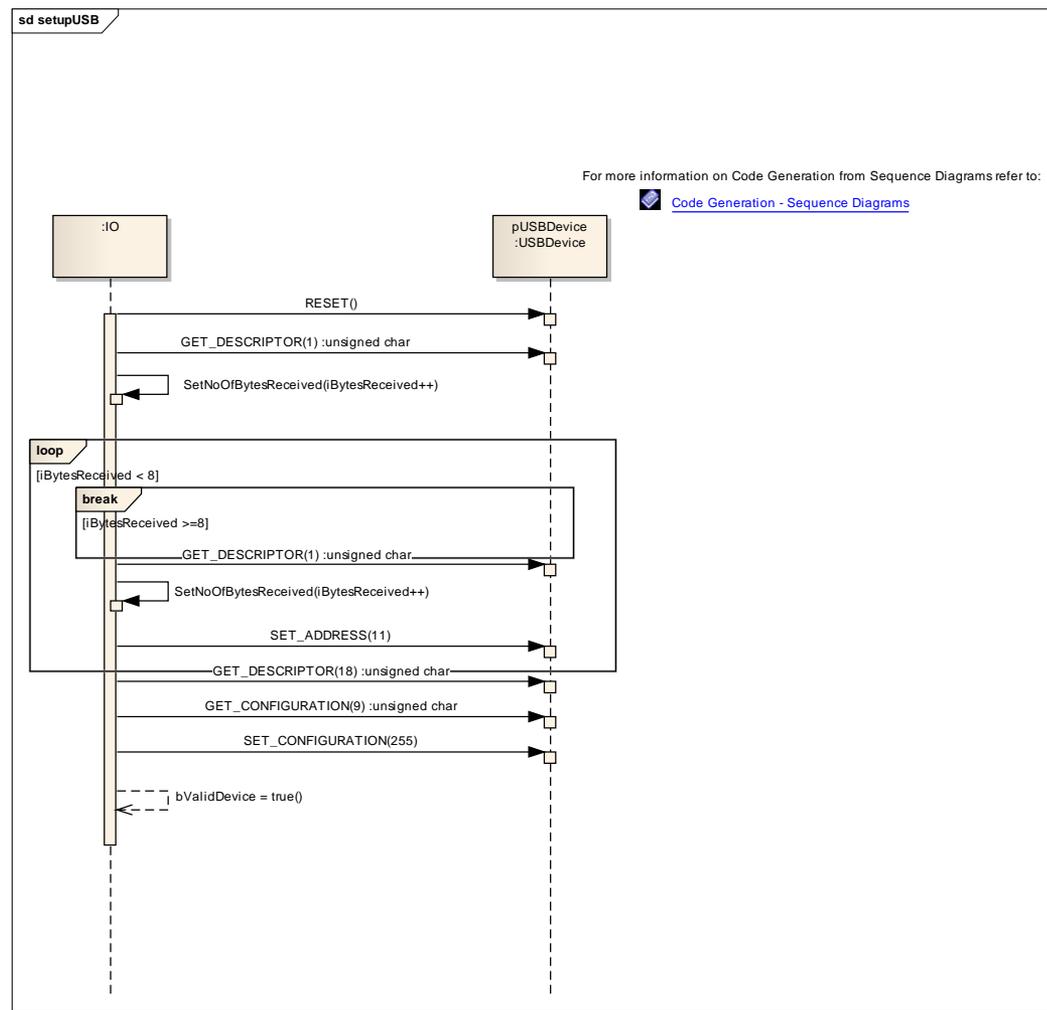


Figure 12 – Sequence diagram for setup USB showing a loop fragment

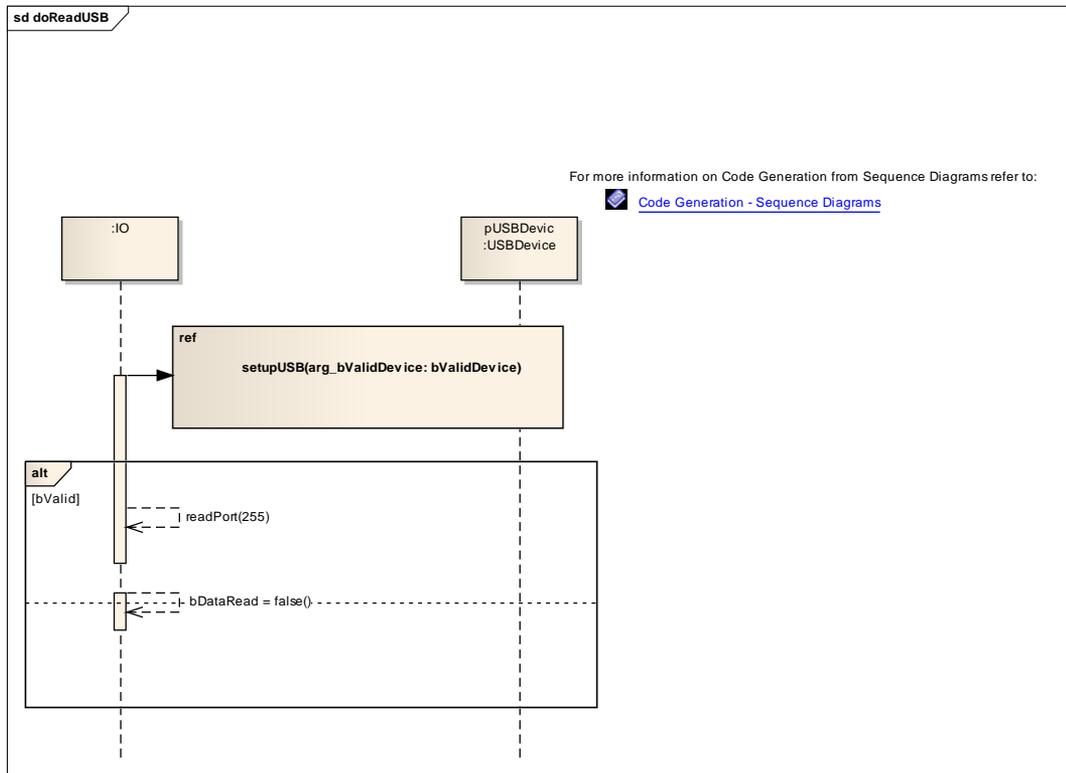


Figure 13 – Reading the USB Port

IO.h

```

73 {
74     TT_NOTTRANSITION
75 };
76 enum CommandType
77 {
78     Do,
79     Entry,
80     Exit
81 };
82 private:
83     StateType currState;
84     StateType nextState;
85     TransitionType currTransition;
86     bool transcend;
87     StateType SearchExternalMedia_history;
88     void searchExternalMedia_ReadSerialPorts(CommandType command);
89
90     void searchExternalMedia_ReadUSBPorts(CommandType command);
91     void StatesProc(StateType currState, CommandType command);
92     void TransitionsProc(TransitionType transition);
93     void initializeStateMachine();
94     void runStateMachine();
95
96     /* End - EA generated code for StateMachine */
97 };
98
99 #endif // !defined(EA_789E55B0_BBD8_4c3d_B0B3_39C99FB6BAB2_INCLUDED_)
100
101
  
```

Figure 14 – Generated C++ code for IO

## Generating VB.Net and Java from State and Activity Diagrams

Hopefully by now you're getting the idea that Enterprise Architect can generate behavioral code in just about any language from state, activity, and sequence diagrams. We'll illustrate this first by showing VB.Net code (Figure 16) for the Search External Media state machine (Figure 15).

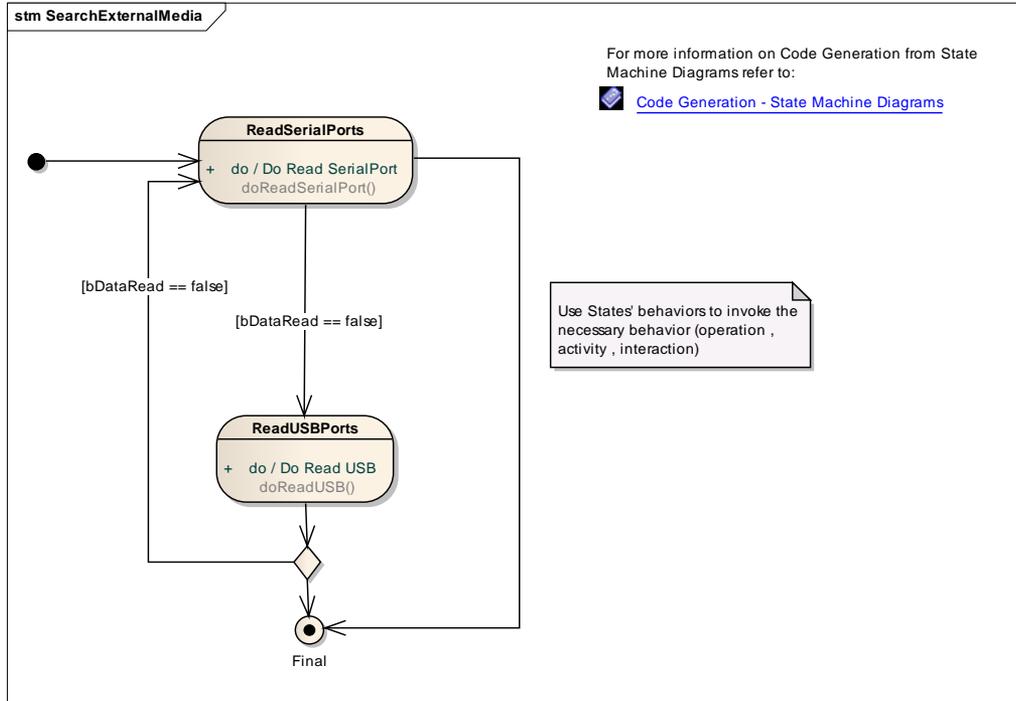


Figure 15 – State Machine for Searching External Media

```

IO.vb
160
161     TT_NOTTRANSITION
162 End Enum
163 Private Enum CommandType
164     BehDo
165     BehEntry
166     BehExit
167 End Enum
168 private currState As StateType
169 private nextState As StateType
170 private currTransition As TransitionType
171 private transcend as Boolean
172 private SearchExternalMedia_history As StateType
173 Private Sub searchExternalMedia_ReadSerialPorts(ByVal comma
174     Select Case command
175         case CommandType.BehDo
176             'Do Behaviors..
177             doReadSerialPort()
178             'State's Transitions
179             Dim bFlag as Boolean
180             If (bDataRead = false) Then
181                 bFlag = true
182                 nextState = StateType.SearchExternalMedia_R
183             End If
184             If ( bFlag = False ) Then
185                 nextState = StateType.ST_NOSTATE'Final Stat
186             End If
187         End Select
188     End Sub

```

Figure 16 – VB.Net behavioral code, automatically generated from the state machine above

Finally, we'll wrap up this section by showing the activity diagram and generated Java code for Read Serial Port in Figures 17 and 18.

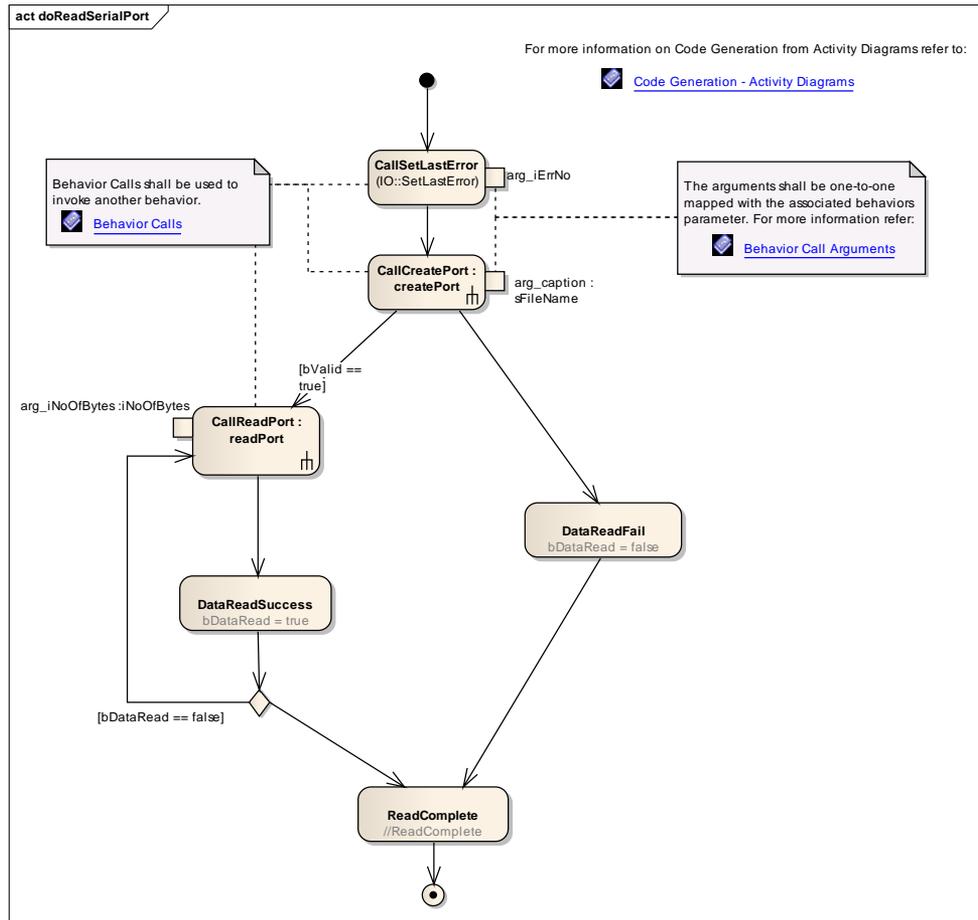


Figure 17 – Activity Diagram for Reading the Serial Port

```

IO.java
123 public void createPort(String sFileName)
124 {
125     // behavior is a Activity
126     //Create a handle to port;
127 }
128
129 public void doReadSerialPort()
130 {
131     // behavior is a Activity
132     SetLastError(0);
133     createPort("LPT1");
134     if (bValidPort)
135     {
136         readPort(1);
137         bDataRead = true;
138         while (bDataRead == false)
139         {
140             readPort(1);
141             bDataRead = true;
142         }
143     }
144     else
145     {
146         bDataRead = false;
147     }
148     //ReadComplete;
149 }
150
151 public void setupUSB(boolean bValidDevice)

```

Figure 18 – Behaviorally generated Java code for Read Serial Port

## Customizing The Code Generator

Enterprise Architect uses a template-driven approach to code generation, and provides an editor for tailoring the code templates.

Enterprise Architect's code templates specify the transformation from UML elements to the various parts of a given programming language. The templates are written as plain text with a syntax that shares some aspects of both mark-up languages and scripting languages.

Figure 19 shows the Code Template Editor being used to tailor how C# code is generated.

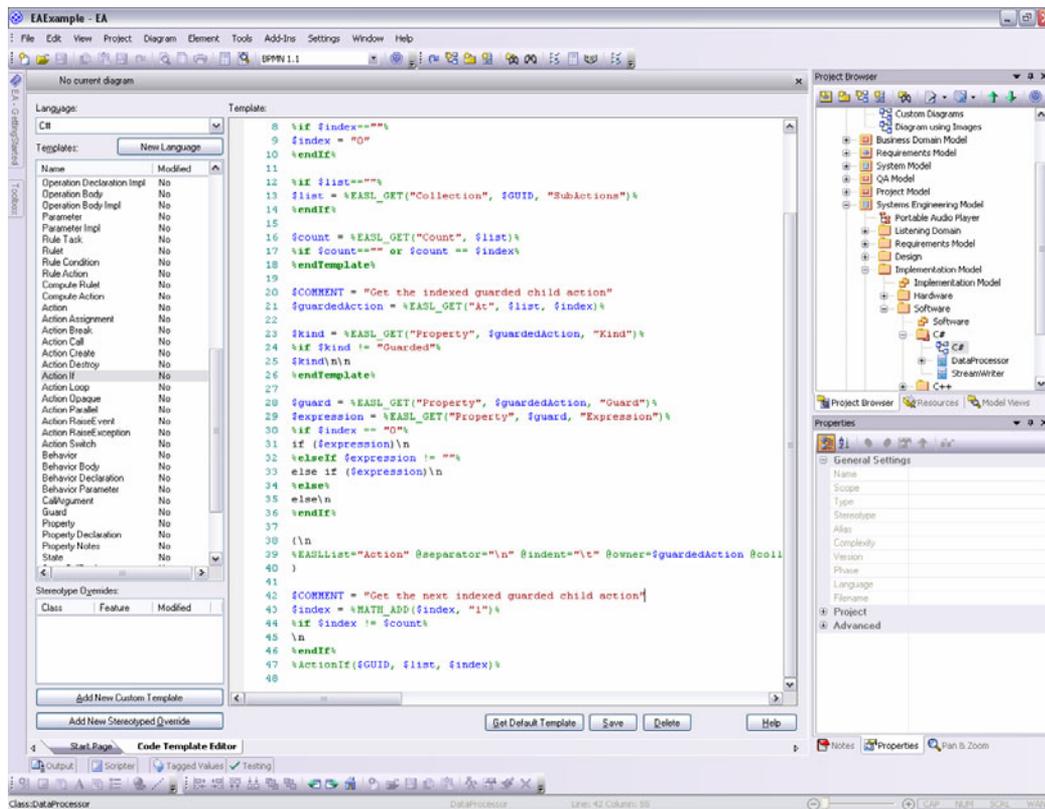


Figure 19 – ActionIf Code template for C#

Code Templates are written as plain text. The template syntax centers on three basic constructs:

- Literal Text
- Macros
- Variables

Templates can contain any or all of these constructs.

## Integrating Models and Code in your favorite IDE

Since the beginning of modeling time, the gap (sometimes a chasm) between models and code has always been problematic. Models, the argument goes, don't represent reality... only the code represents reality... therefore the model must be worthless, and we should just skip modeling and jump straight to code.

Those who have used this argument to avoid modeling probably felt quite safe in doing so because nobody has ever managed to make "reverse engineering" or "round-trip engineering" a seamless process... until now. But that's exactly the problem that the MDG Integration technology (available for both Visual Studio and Eclipse) from Sparx Systems solves.

So... here's the six million dollar question: how do we keep the model and the code synchronized over the lifetime of the project? You can see the answer in Figure 20.

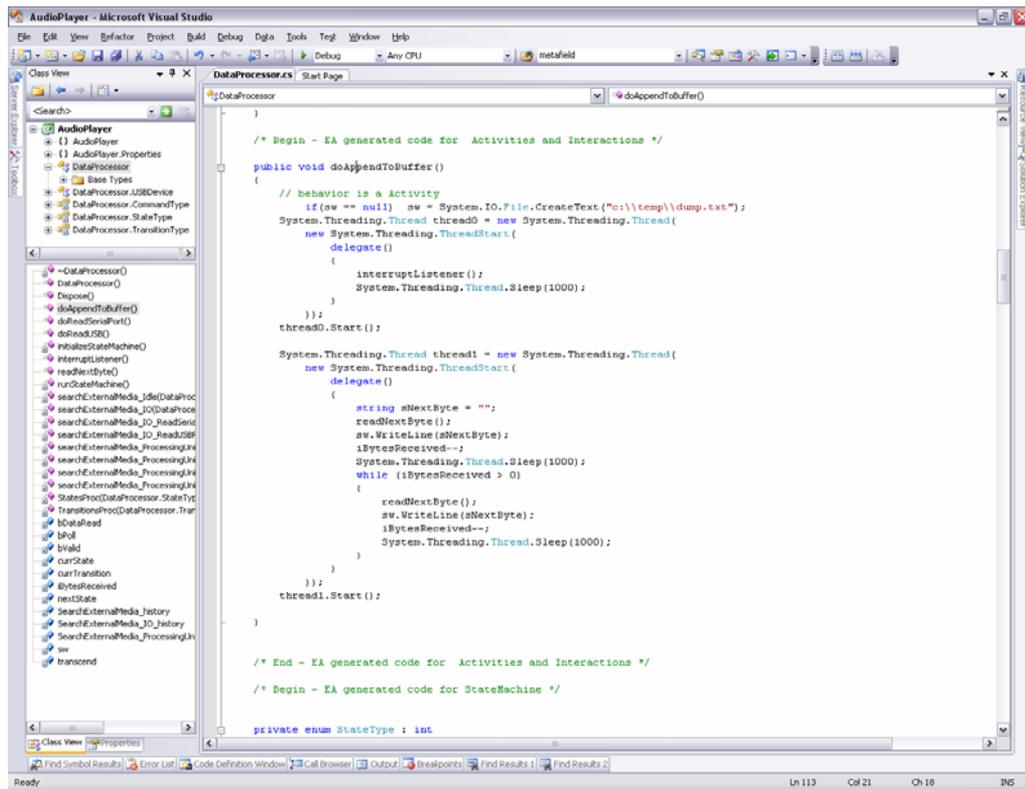


Figure 20 – Generated C# code for DataProcessor in Microsoft Visual Studio 2008

Here's how it works:

- 1) Connect your UML model to a Visual Studio or Eclipse Project
- 2) Link a package in your model to the project in your IDE
- 3) Browse the source code by clicking on operations on the classes
- 4) Edit the source code in your IDE

MDG Integration keeps the model and code in-synch for you. Problem solved.

## **Wrapping up**

That concludes our roadmap for embedded systems development using SysML and the Enterprise Architect System Engineering Edition. Our roadmap has taken us through Requirements definition, allocation, and traceability, hardware and software design, constraints and parametrics, and through implementation using behavioral code generation for software and for hardware using hardware-description languages.

---

We hope you've found this eBook useful. You can contact ICONIX with comments or questions at [SysMLTraining@iconixsw.com](mailto:SysMLTraining@iconixsw.com), or explore our "[SysML JumpStart Training](#)" on our website.

**We wish you success in your development efforts!**

---