An Illustrated Example using Enterprise Architect

Embedded Systems Development using SysML

By Doug Rosenberg
with Sam Mancarella

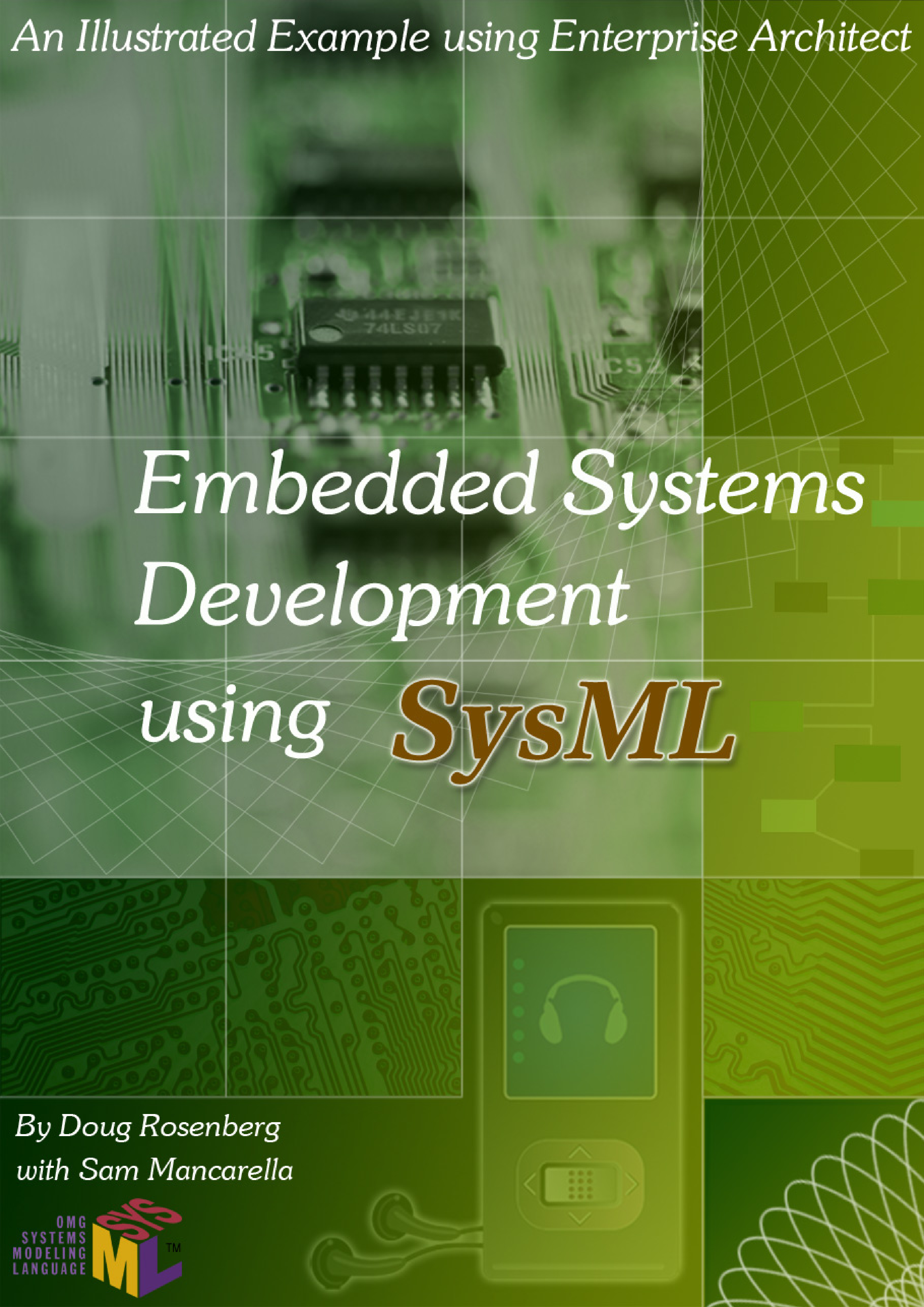


Table of Contents

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[1]) 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.

[1] 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***.

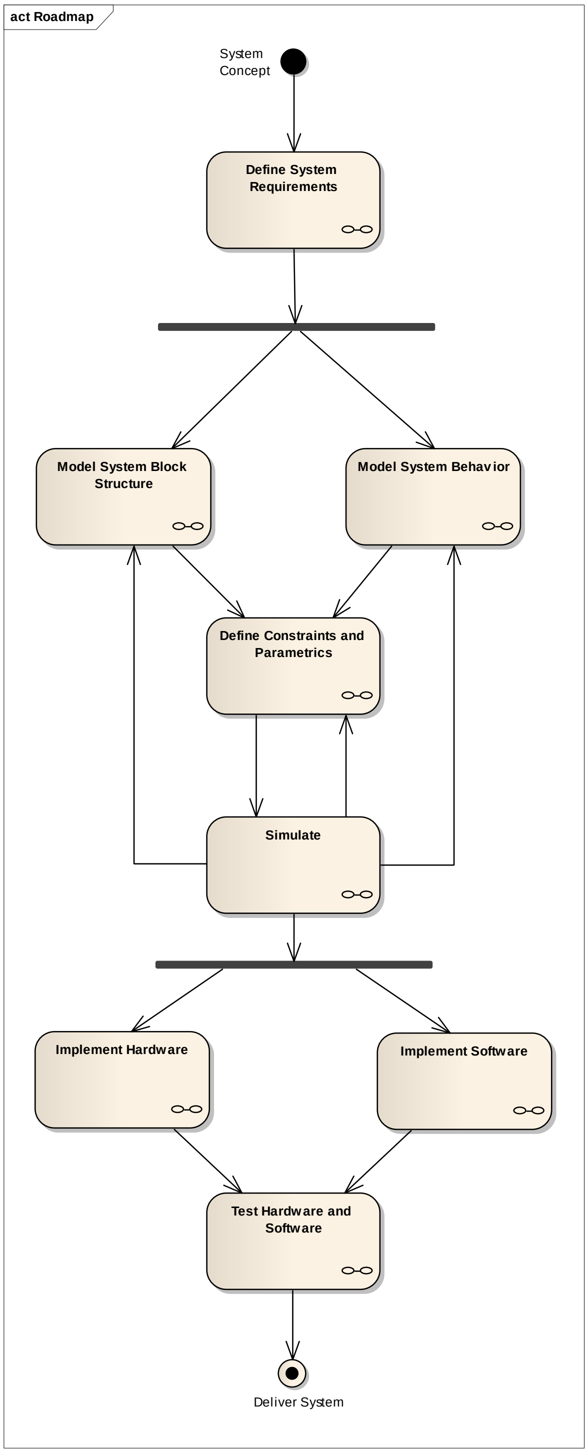

act Roadmap
System Concept
Define System Requirements
Model System Block Structure
Model System Behavior
Define Constraints and Parametrics
Simulate
Implement Hardware
Implement Software
Test Hardware and Software
Deliver System

Figure 1 – ICONIX 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".[2]

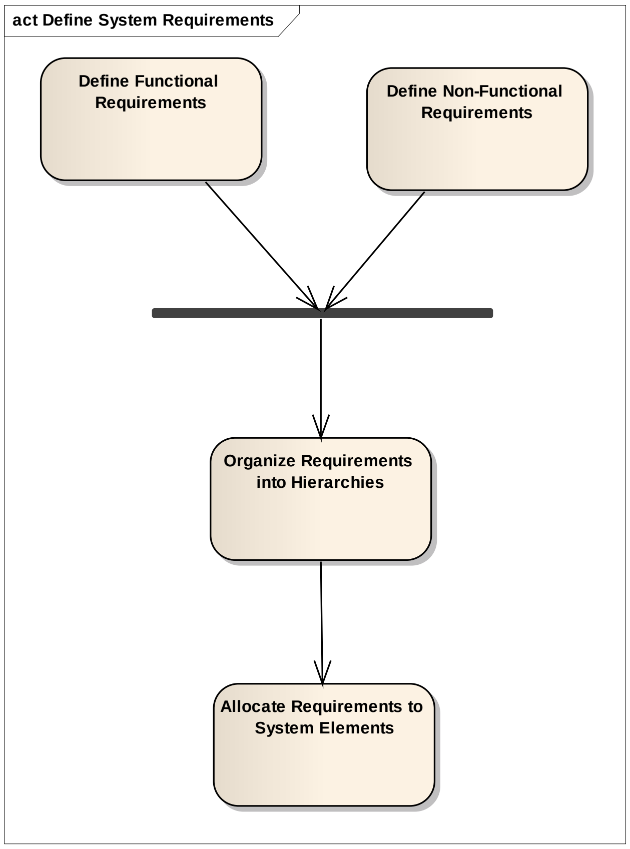

act Define System Requirements
Define Functional Requirements
Define Non-Functional Requirements
Organize Requirements into Hierarchies
Allocate Requirements to System Elements

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.

[2] OMG Systems Modeling Language Tutorial, INCOSE 2008

Structure

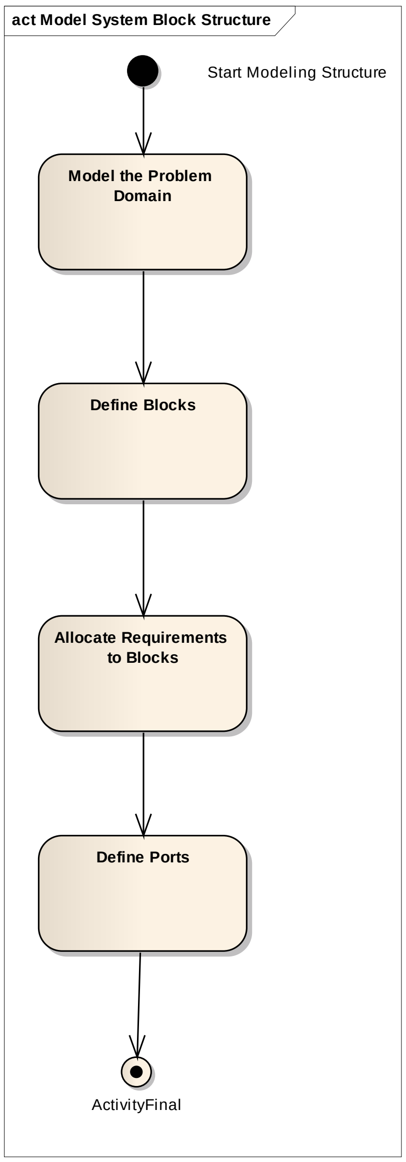


Figure 3 – Roadmap: Model System Block Structure

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

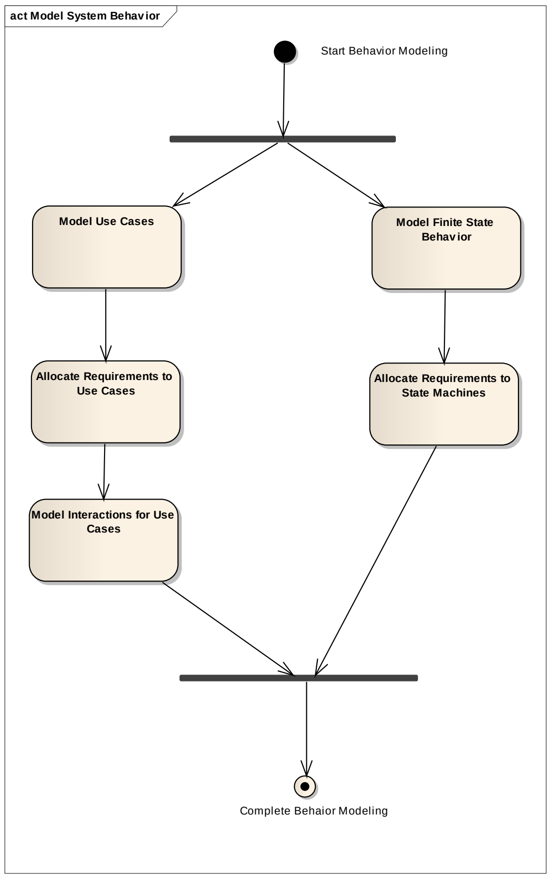

act Model System Behavior
Start Behavior Modeling
Model Use Cases
Model Finite State Behavior
Allocate Requirements to Use Cases
Allocate Requirements to State Machines
Model Interactions for Use Cases
Complete Behaior Modeling

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[3], 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.

[3] 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

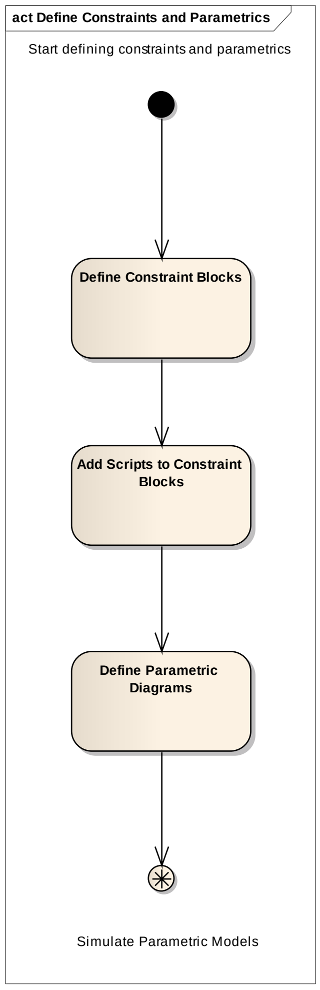

act Define Constraints and Parametrics
Start defining constraints and parametrics
Define Constraint Blocks
Add Scripts to Constraint Blocks
Define Parametric Diagrams
Simulate Parametric Models

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

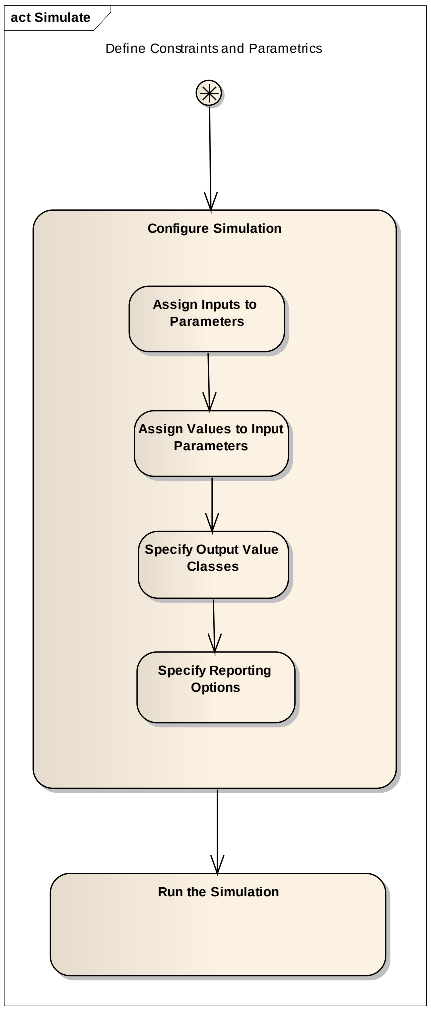

act Simulate
Define Constraints and Parametrics
Configure Simulation
Assign Inputs to Parameters
Assign Values to Input Parameters
Specify Output Value Classes
Specify Reporting Options
Run the Simulation

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[4]:

> *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.*

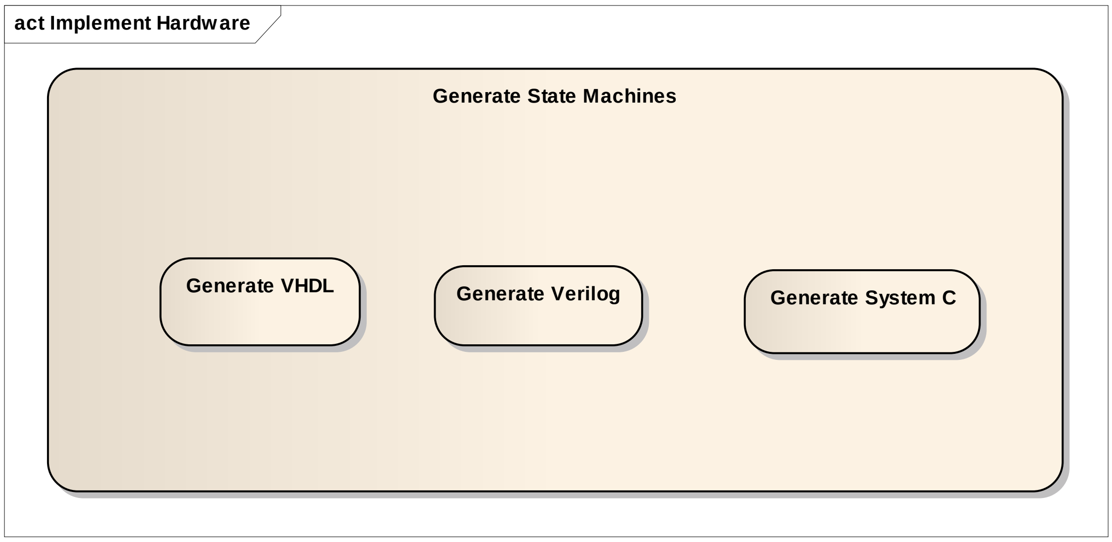


Figure 7 – Roadmap: Implement Hardware

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.

[4] 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.

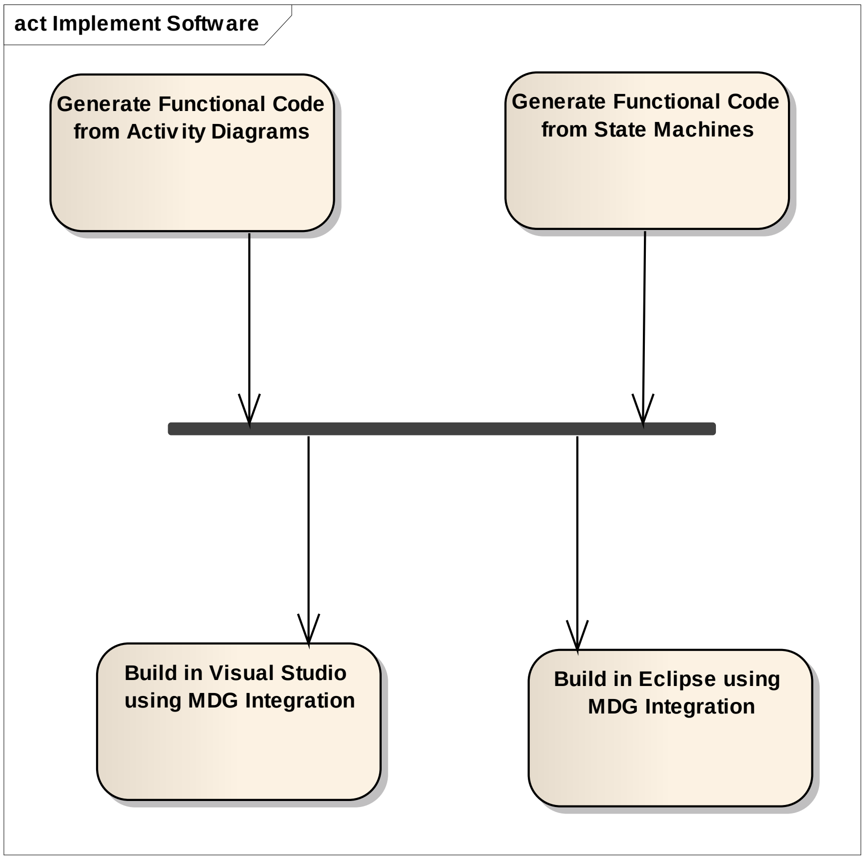

act Implement Software
Generate Functional Code from Activity Diagrams
Generate Functional Code from State Machines
Build in Visual Studio using MDG Integration
Build in Eclipse using MDG Integration

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.

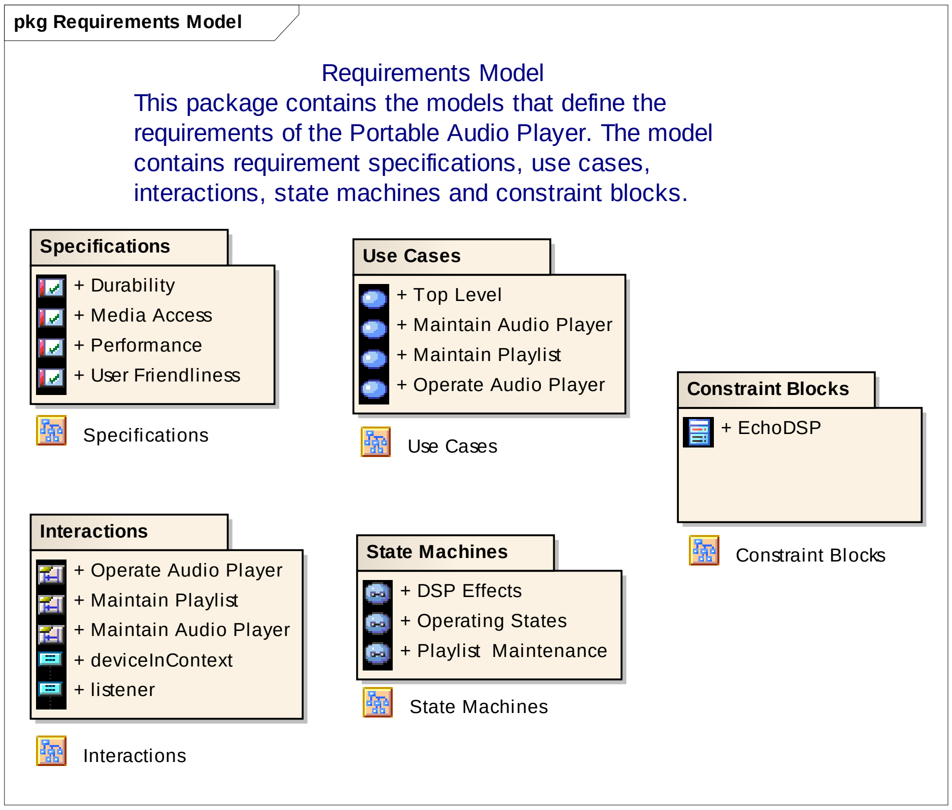


Figure 9 – SysML models are organized into Requirements, Behavior, Structure, Constraints and Parametrics, and include both Hardware and Software Implementation models.

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).

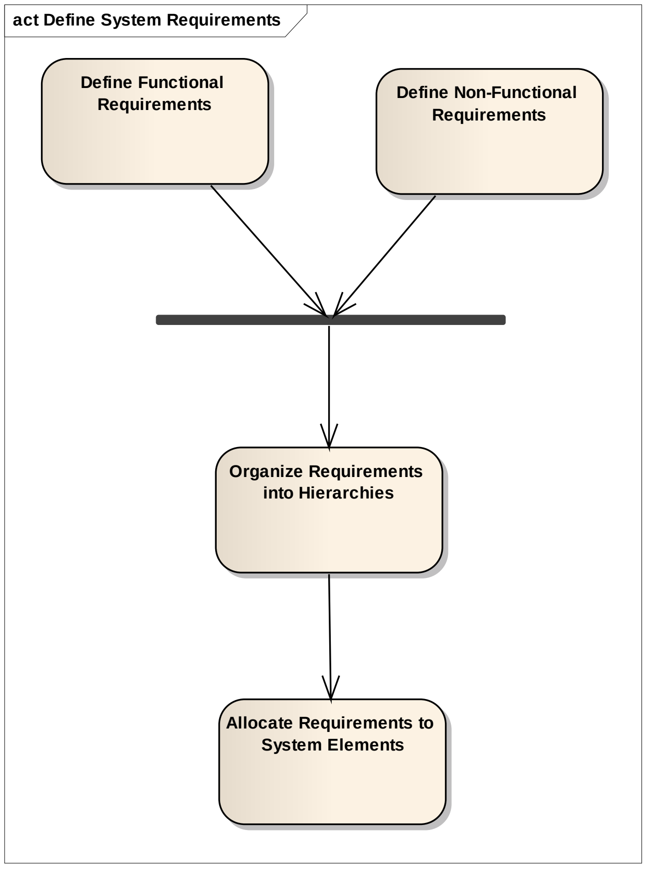

act Define System Requirements
Define Functional Requirements
Define Non-Functional Requirements
Organize Requirements into Hierarchies
Allocate Requirements to System Elements

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**[5].

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.

[5] 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’.

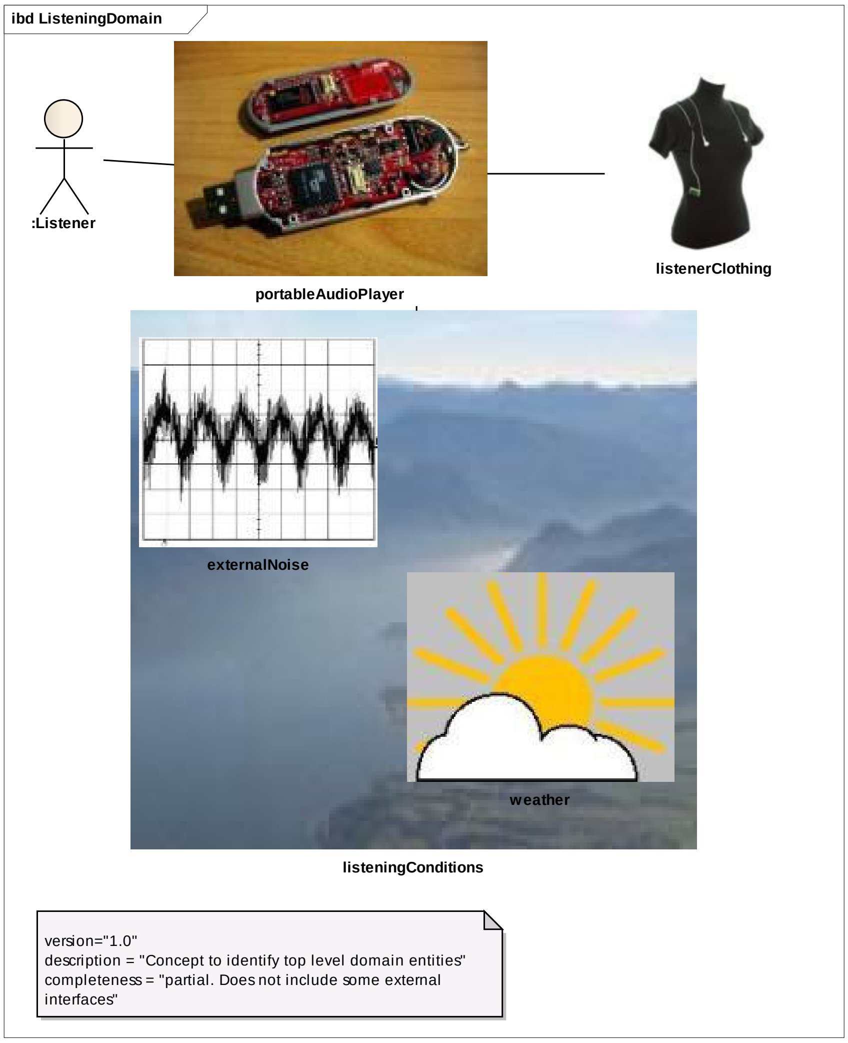

ibd ListeningDomain
:Listener
portableAudioPlayer
listenerClothing
externalNoise
weather
listeningConditions
version="1.0"
description = "Concept to identify top level domain entities"
completeness = "partial. Does not include some external interfaces"

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

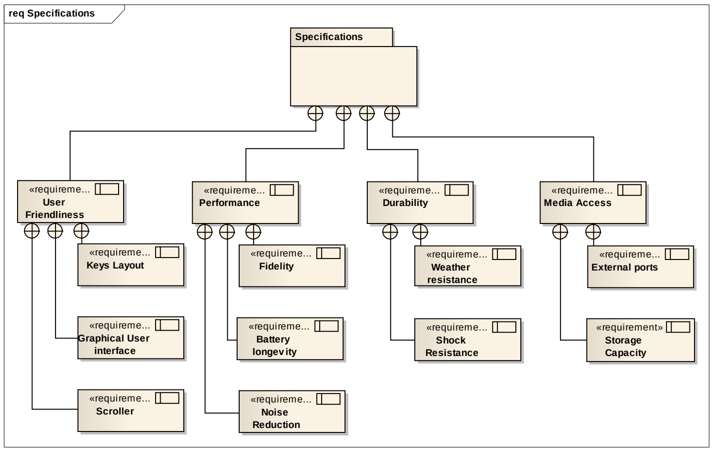
Modeling Tip – It’s easy to import graphics into Enterprise Architect Models

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*.

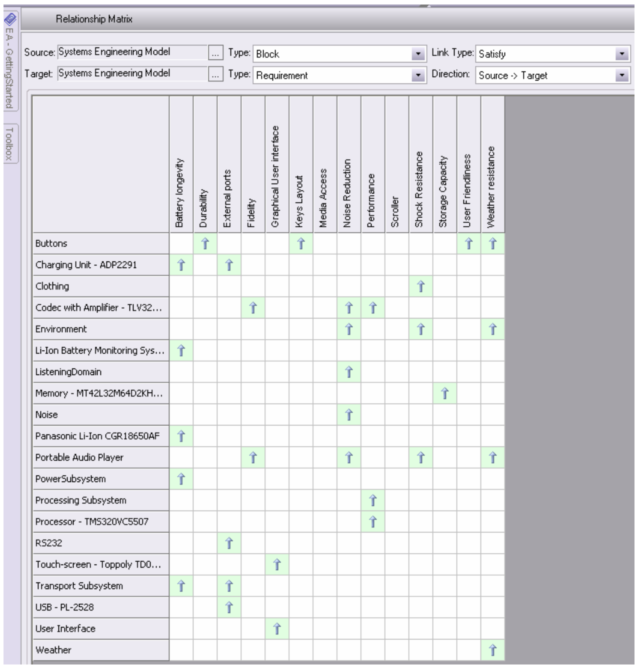
The "crosshair" notation in the Audio Player Requirements diagram below shows the use of *containment* to organize requirements hierarchically into Categories meaning that the Specifications Package OWNS the requirements, which in turn, own the 'sub requirements' beneath.



req Specifications
Specifications
«requireme... User Friendliness
«requireme... Performance
«requireme... Durability
«requireme... Media Access
«requireme... Keys Layout
«requireme... Graphical User interface
«requireme... Scroller
«requireme... Fidelity
«requireme... Battery longevity
«requireme... Noise Reduction
«requireme... Weather resistance
«requireme... Shock Resistance
«requireme... External ports
«requirement» Storage Capacity

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.



Relationship Matrix

Source: Systems Engineering Model Type: Block Link Type: Satisfy
Target: Systems Engineering Model Type: Requirement Direction: Source -> Target

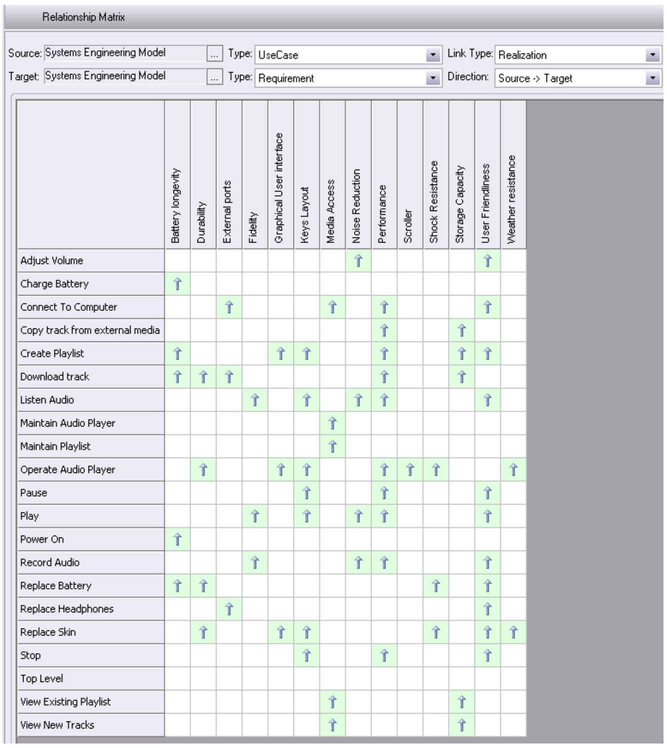
	Battery longevity	Durability	External ports	Fidelity	Graphical User interface	Keys Layout	Media Access	Noise Reduction	Performance	Scroller	Shock Resistance	Storage Capacity	User Friendliness	Weather resistance
Buttons		↑				↑							↑	↑
Charging Unit - ADP2291	↑		↑											
Clothing											↑			
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 - TMS320VC5507									↑					
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.


Relationship Matrix

Source: Systems Engineering Model … Type: UseCase Link Type: Realization
Target: Systems Engineering Model … Type: Requirement Direction: Source -> Target

	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.

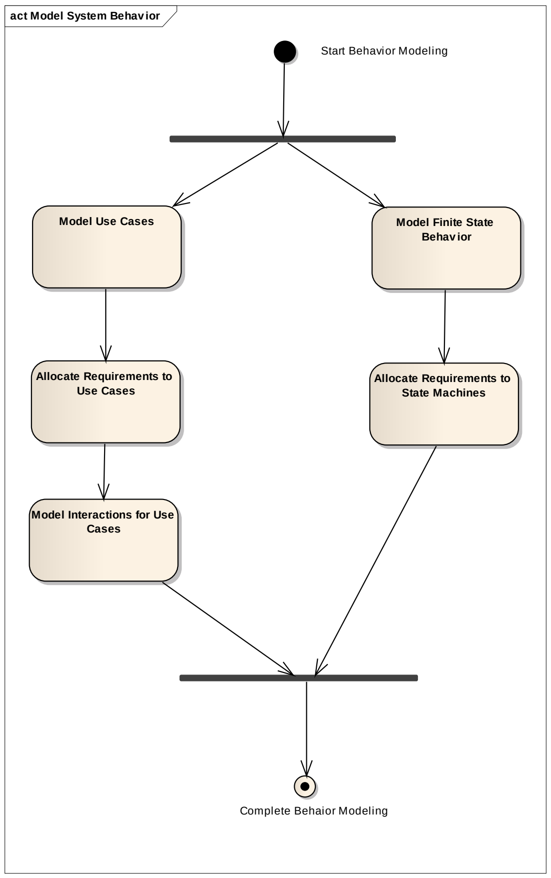

act Model System Behavior
Start Behavior Modeling
Model Use Cases
Model Finite State Behavior
Allocate Requirements to Use Cases
Allocate Requirements to State Machines
Model Interactions for Use Cases
Complete Behaior Modeling

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.

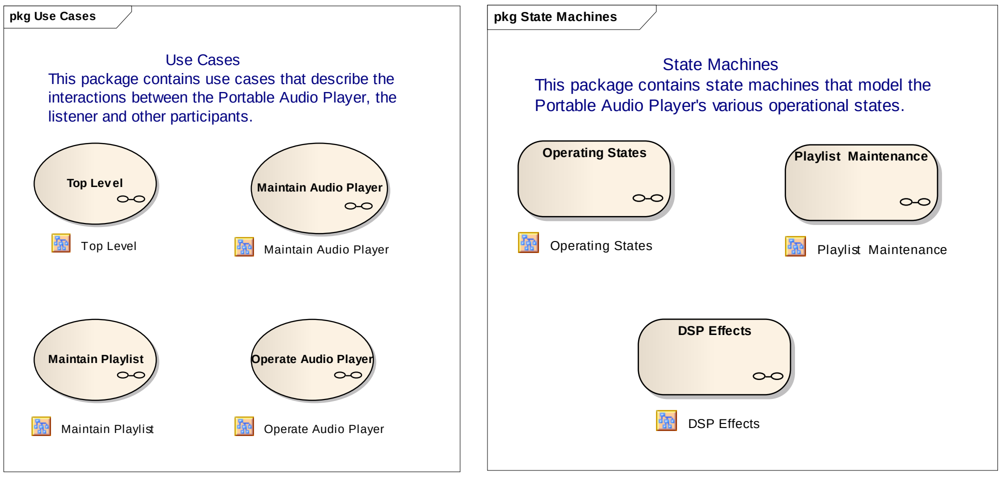


Figure 2 – Behavioral models include Use Cases, Interactions, and State Machines

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.

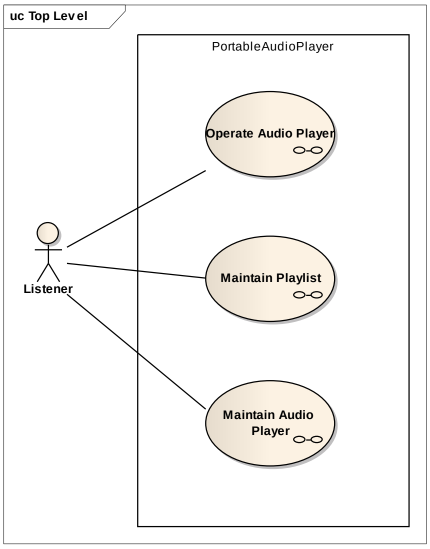


Figure 3 – Audio Player Top Level Use Cases

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.

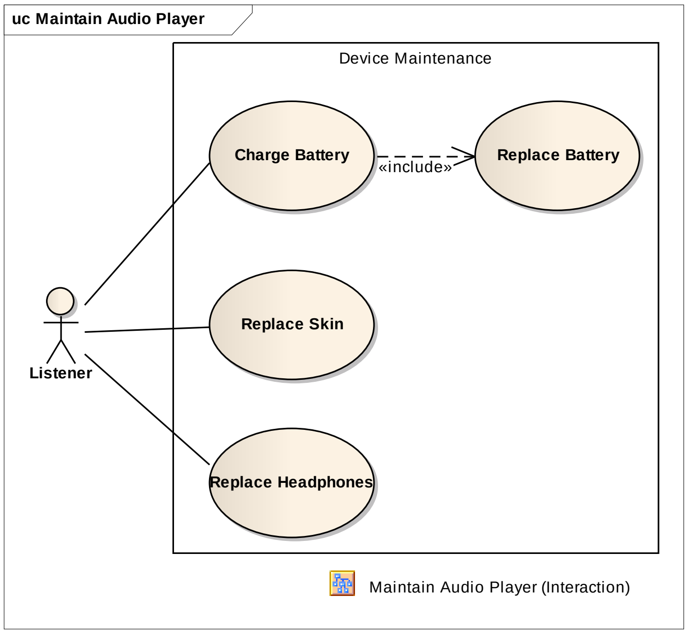


Figure 4 – Use cases for maintaining the audio player hardware include charging and replacing the battery, and replacing the skin and the headphones.

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.

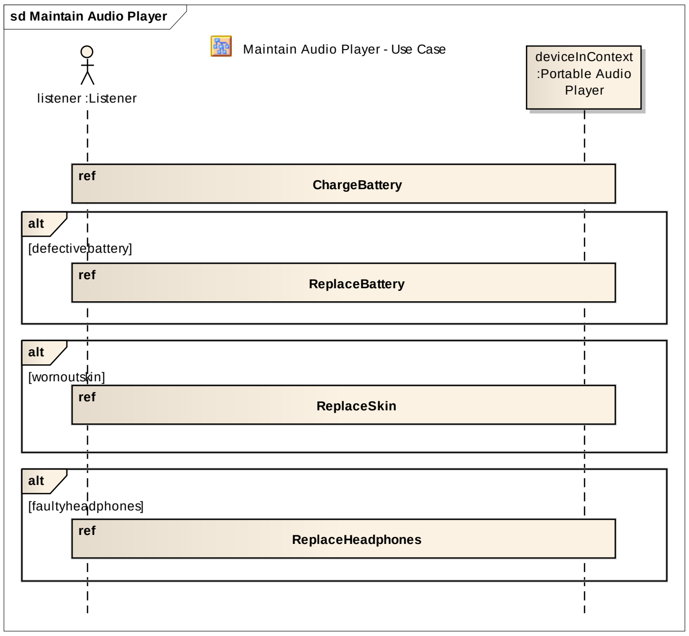


Figure 5 – The interaction diagram for maintaining the audio player shows 3 alternate courses of action (defective batteries, worn out skin, and faulty headphones) and one normal course (charging the battery).

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

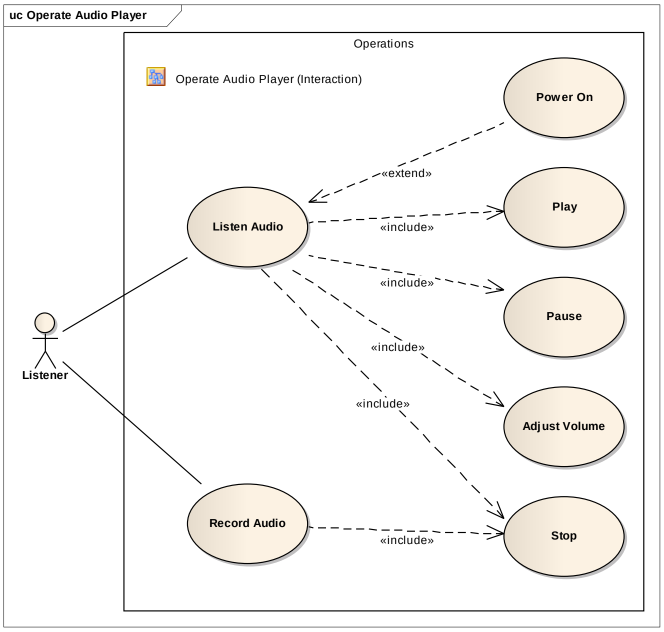


Figure 6 – Audio Player Use Cases for Listening and Recording

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.

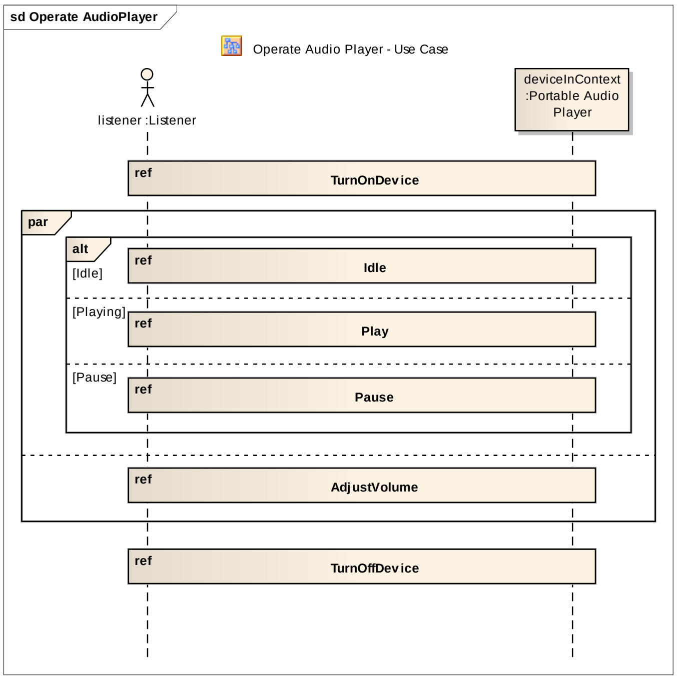


Figure 7 – Audio Player Interaction diagram for Listening/Recording

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.

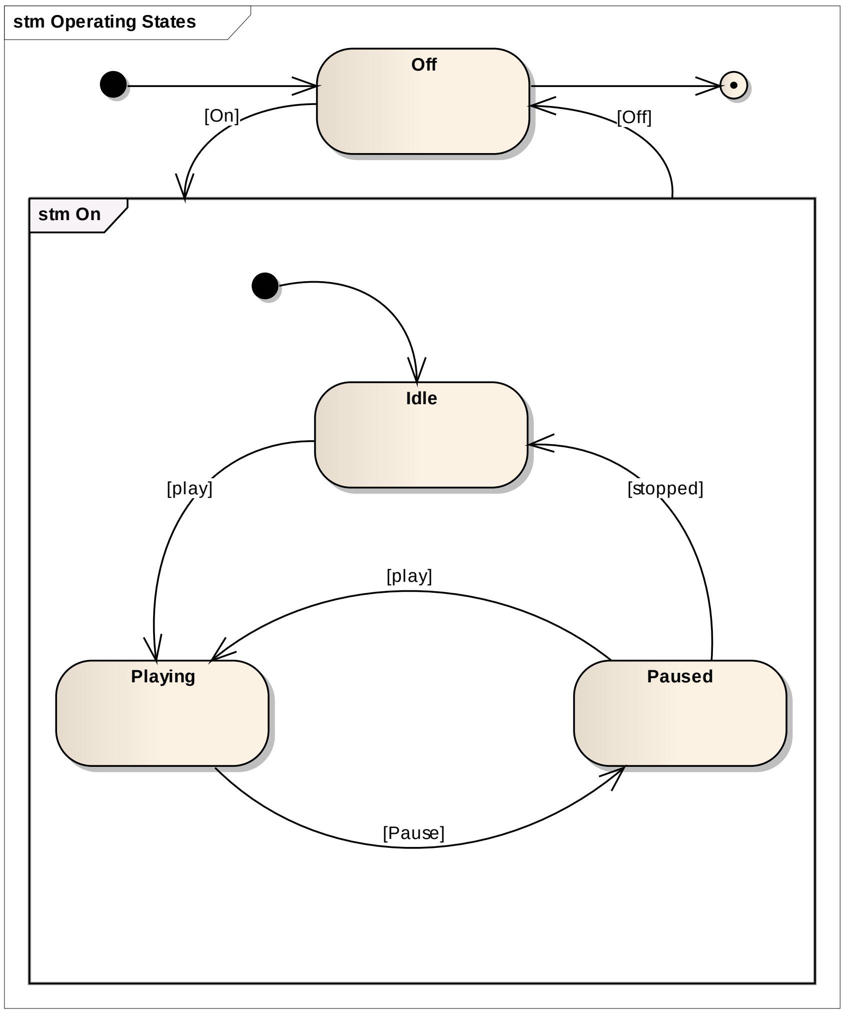


Figure 8 – Audio Player Operating States

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.

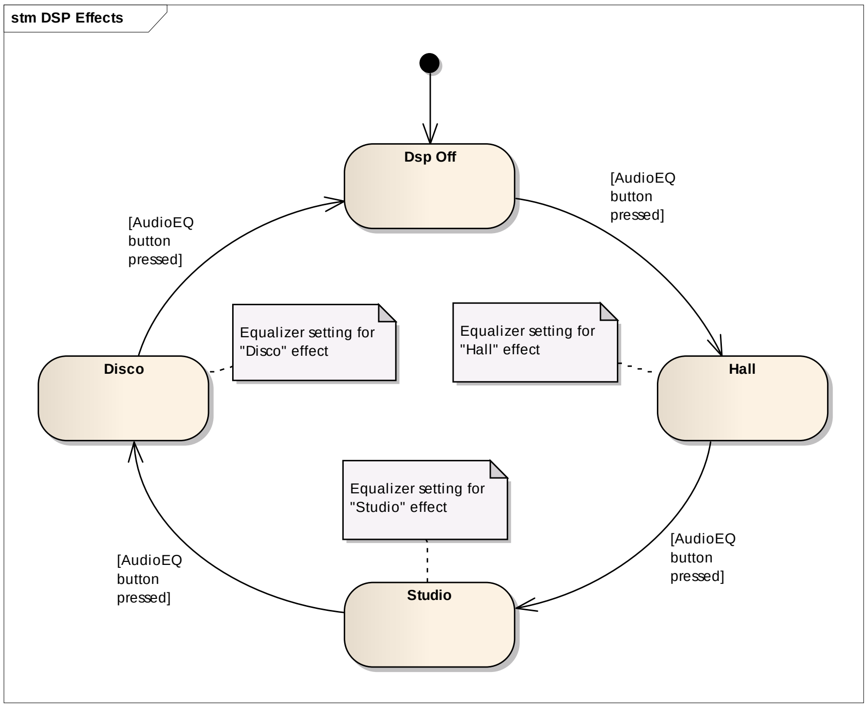


Figure 9 – State Machine for Digital Signal Processing Audio Effects

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.

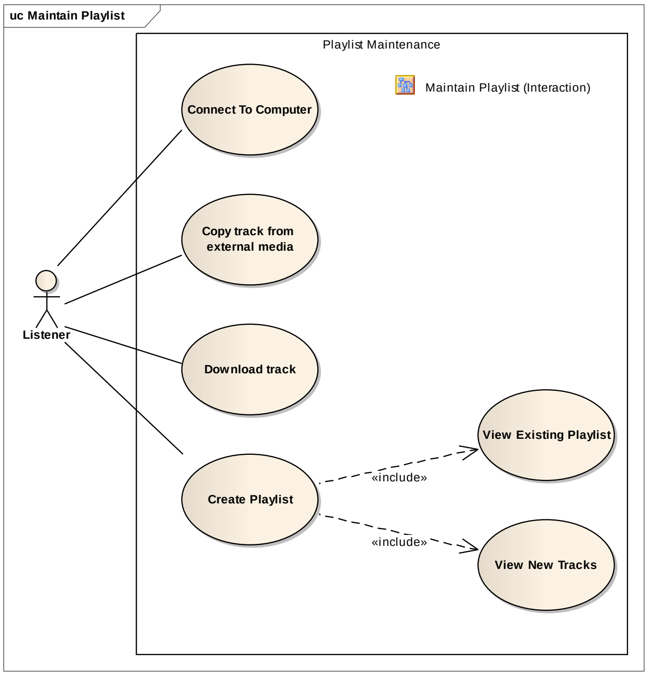


Figure 10 – Use Case Diagram for Playlist Maintenance

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.

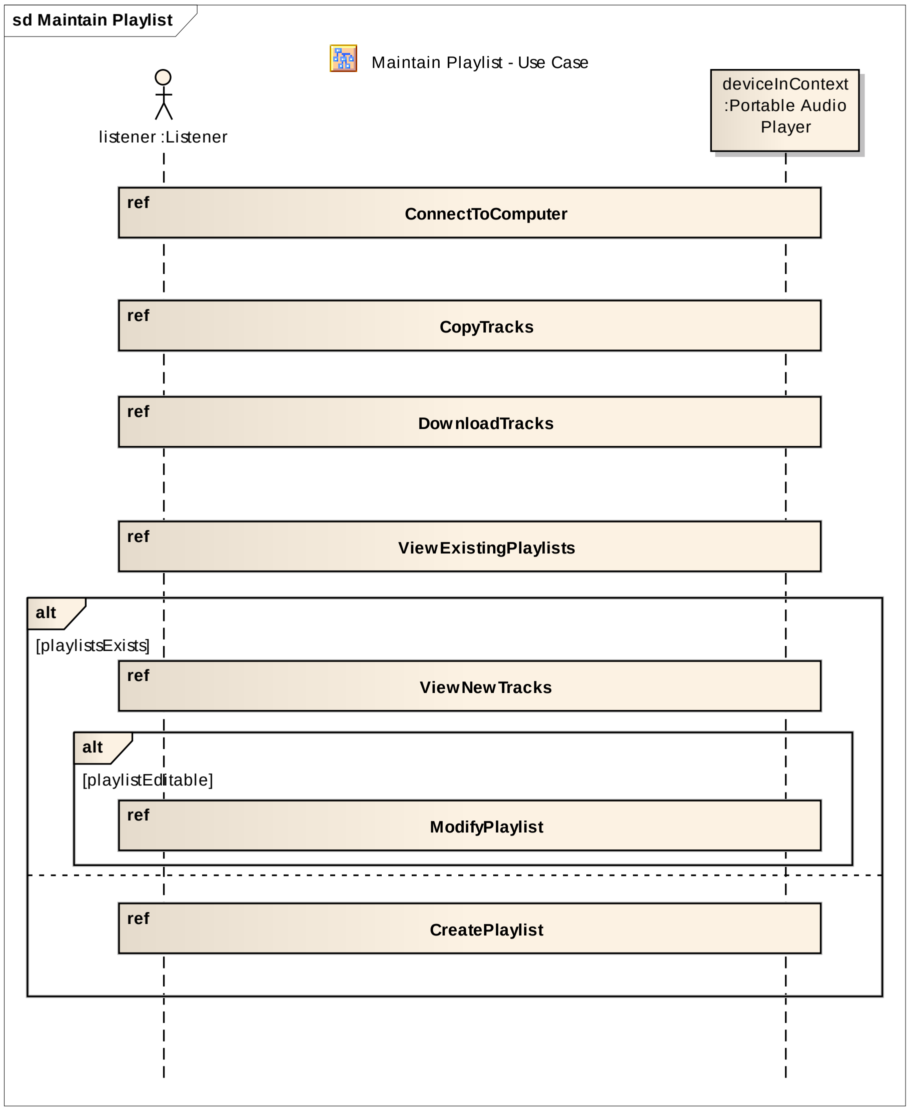


Figure 11 – Scenarios for maintaining playlists

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.

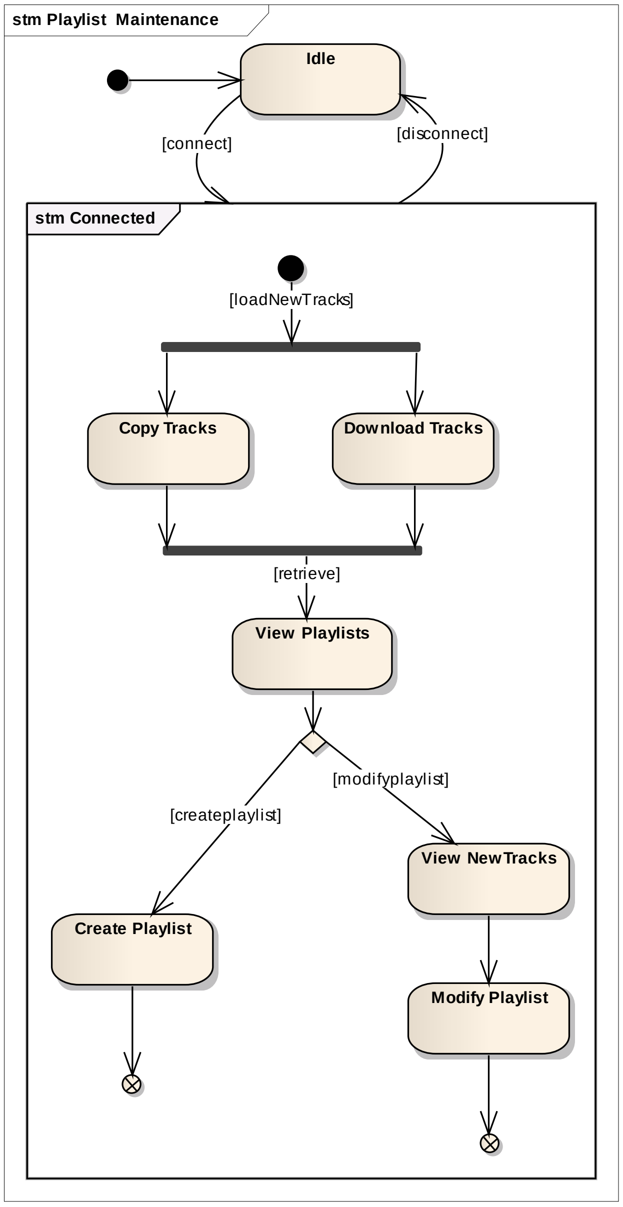


Figure 12 – State/Event behavior for Playlist maintenance

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.

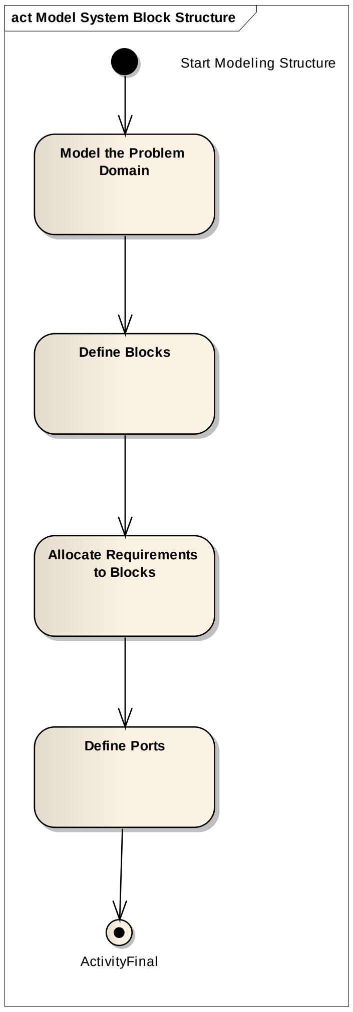


Figure 1: Roadmap - Model Structure

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.

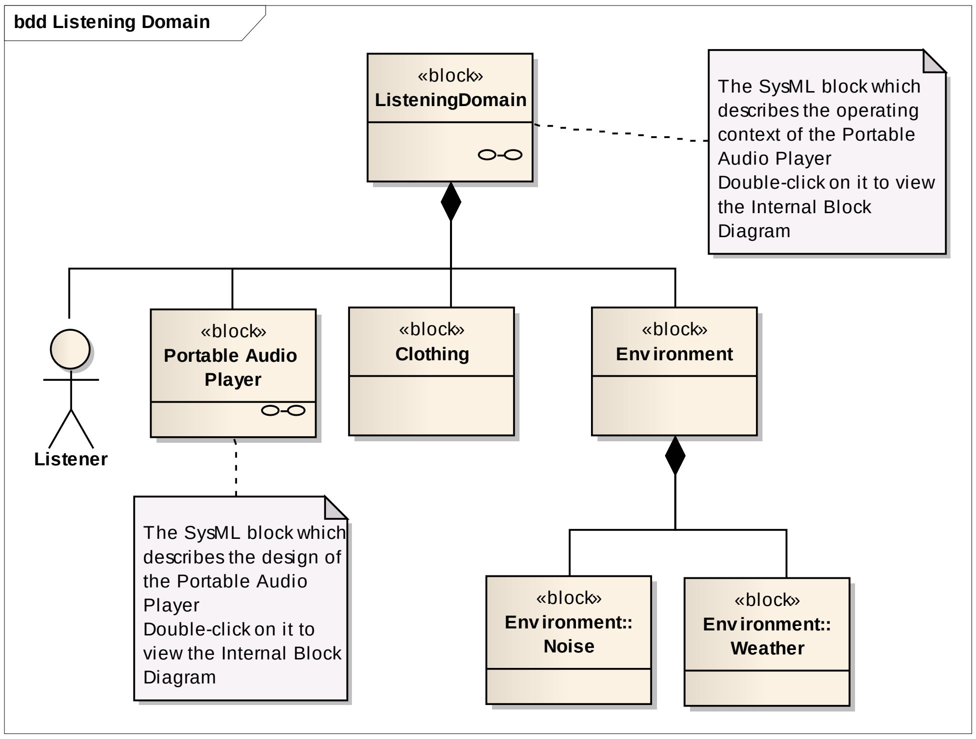

bdd Listening Domain
«block» ListeningDomain
The SysML block which describes the operating context of the Portable Audio Player Double-click on it to view the Internal Block Diagram
Listener
«block» Portable Audio Player
«block» Clothing
«block» Environment
The SysML block which describes the design of the Portable Audio Player Double-click on it to view the Internal Block Diagram
«block» Environment:: Noise
«block» Environment:: Weather

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.

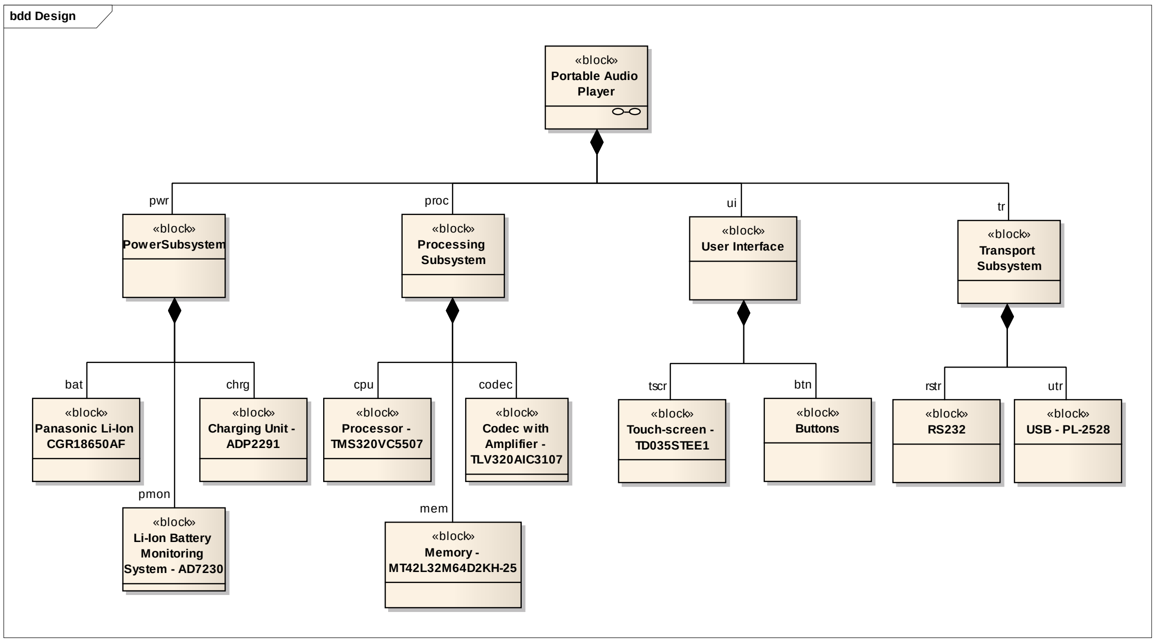

bdd Design
«block» Portable Audio Player
pwr
«block» PowerSubsystem
proc
«block» Processing Subsystem
ui
«block» User Interface
tr
«block» Transport Subsystem
bat
«block» Panasonic Li-Ion CGR18650AF
chrg
«block» Charging Unit - ADP2291
pmon
«block» Li-Ion Battery Monitoring System - AD7230
cpu
«block» Processor - TMS320VC5507
codec
«block» Codec with Amplifier - TLV320AIC3107
mem
«block» Memory - MT42L32M64D2KH-25
tscr
«block» Touch-screen - TD035STEE1
btn
«block» Buttons
rstr
«block» RS232
utr
«block» USB - PL-2528

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.

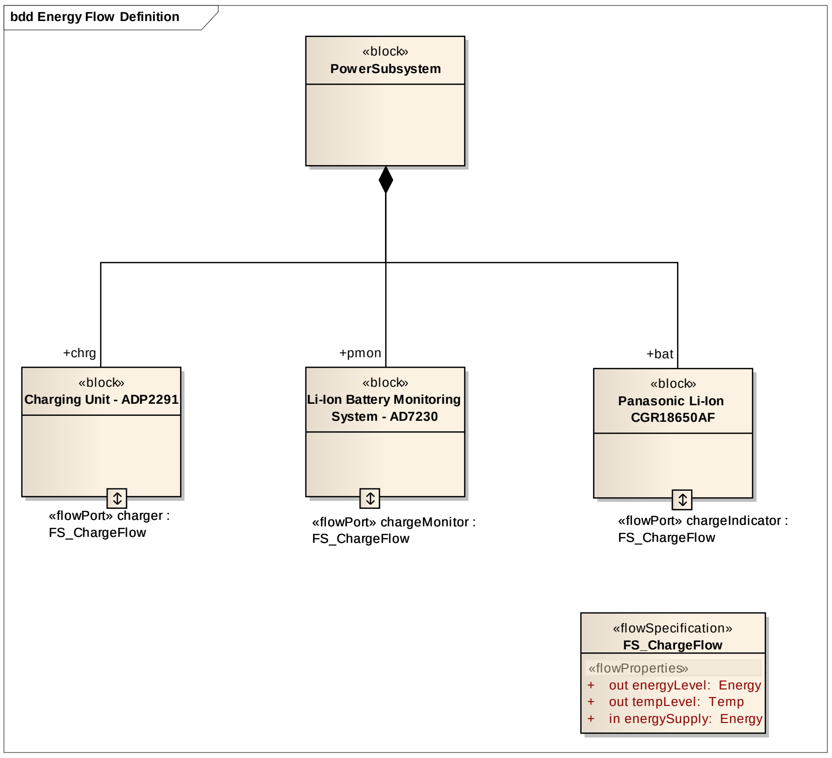

bdd Energy Flow Definition
«block» PowerSubsystem
+chrg
«block» Charging Unit - ADP2291
«flowPort» charger : FS_ChargeFlow
+pmon
«block» Li-Ion Battery Monitoring System - AD7230
«flowPort» chargeMonitor : FS_ChargeFlow
+bat
«block» Panasonic Li-Ion CGR18650AF
«flowPort» chargeIndicator : FS_ChargeFlow
«flowSpecification» FS_ChargeFlow
«flowProperties»
+ out energyLevel: Energy
+ out tempLevel: Temp
+ in energySupply: Energy

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.

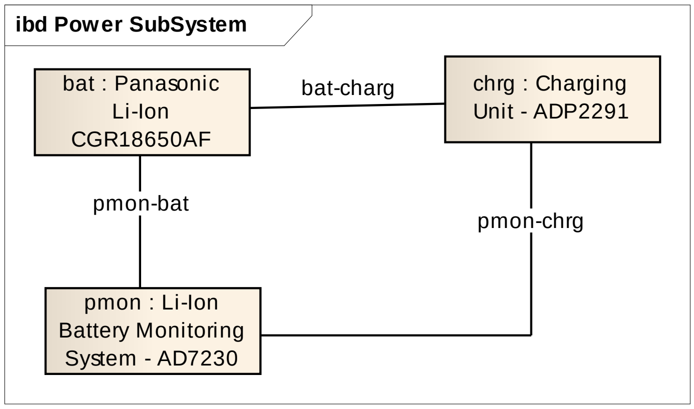

ibd Power SubSystem
bat : Panasonic Li-Ion CGR18650AF
bat-charg
chrg : Charging Unit - ADP2291
pmon-bat
pmon-chrg
pmon : Li-Ion Battery Monitoring System - AD7230

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.

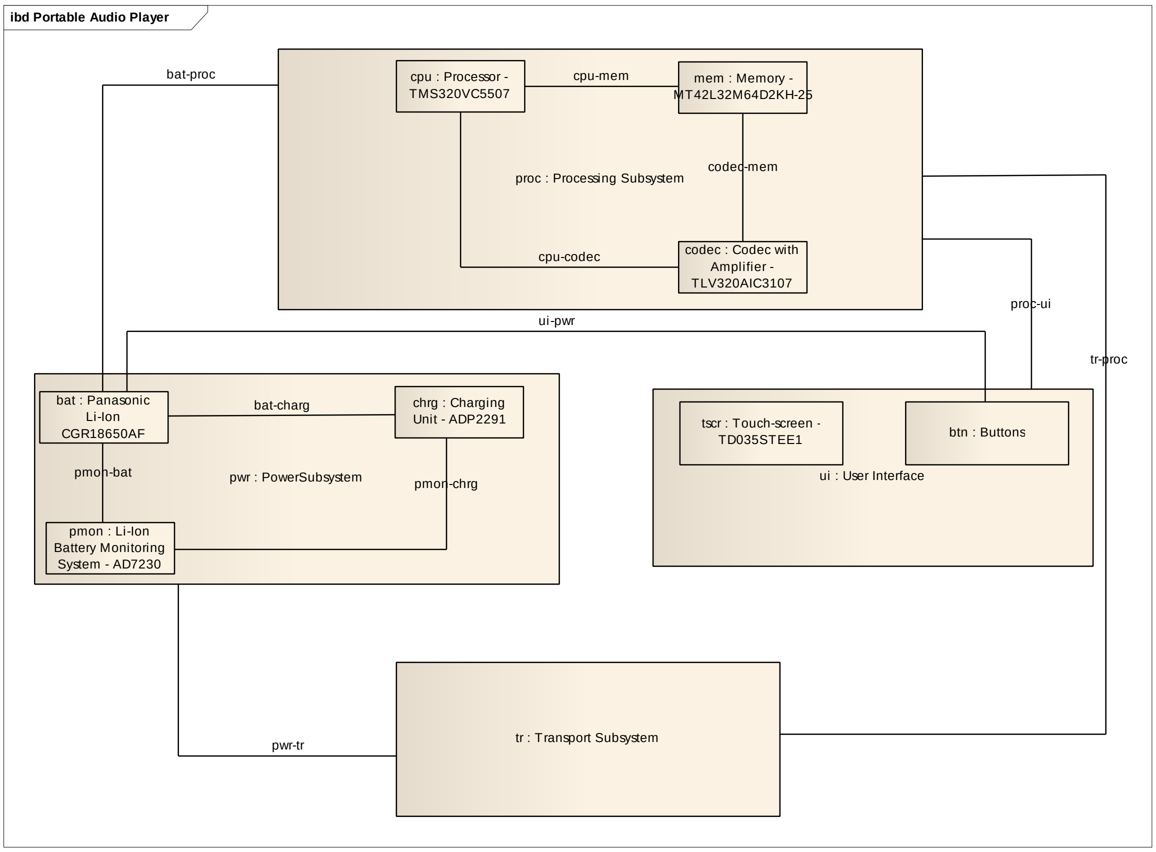

ibd Portable Audio Player
bat-proc
cpu : Processor - TMS320VC5507
cpu-mem
mem : Memory - MT42L32M64D2KH-25
codec-mem
proc : Processing Subsystem
cpu-codec
codec : Codec with Amplifier - TLV320AIC3107
ui-pwr
proc-ui
tr-proc
bat : Panasonic Li-Ion CGR18650AF
bat-charg
chrg : Charging Unit - ADP2291
pmon-bat
pwr : PowerSubsystem
pmon-chrg
pmon : Li-Ion Battery Monitoring System - AD7230
tscr : Touch-screen - TD035STEE1
btn : Buttons
ui : User Interface
tr : Transport Subsystem
pwr-tr

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.

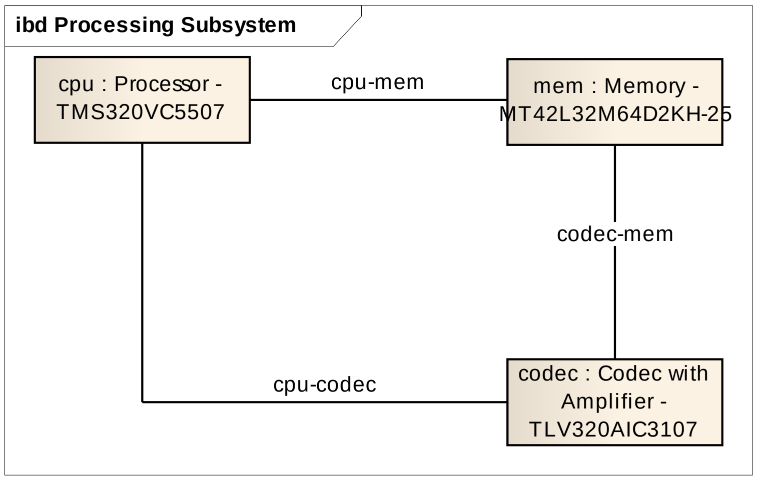

ibd Processing Subsystem
cpu : Processor - TMS320VC5507
cpu-mem
mem : Memory - MT42L32M64D2KH-25
codec-mem
cpu-codec
codec : Codec with Amplifier - TLV320AIC3107

Figure 7 – Audio Player Processing Subsytem 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.

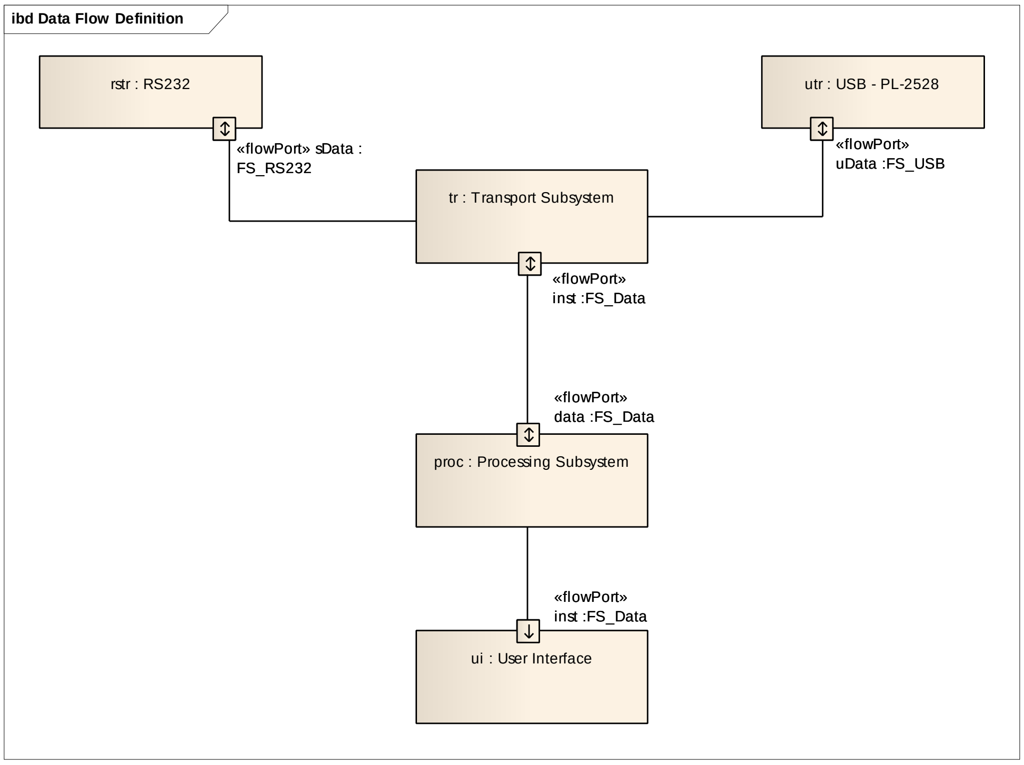


Figure 8 – Audio Player Dataflow between Subsystems

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.

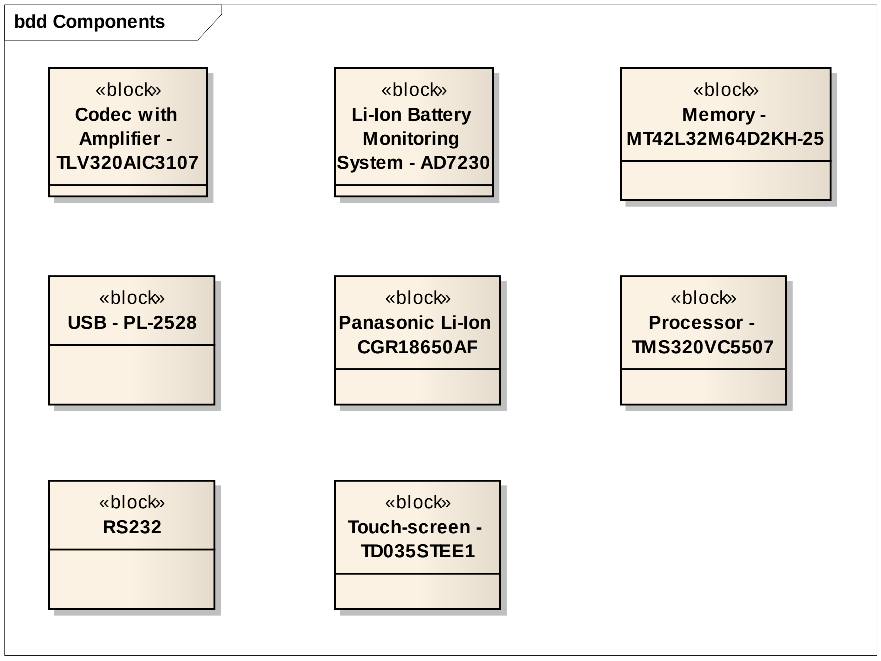

bdd Components
«block» Codec with Amplifier - TLV320AIC3107
«block» Li-Ion Battery Monitoring System - AD7230
«block» Memory - MT42L32M64D2KH-25
«block» USB - PL-2528
«block» Panasonic Li-Ion CGR18650AF
«block» Processor - TMS320VC5507
«block» RS232
«block» Touch-screen - TD035STEE1

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.

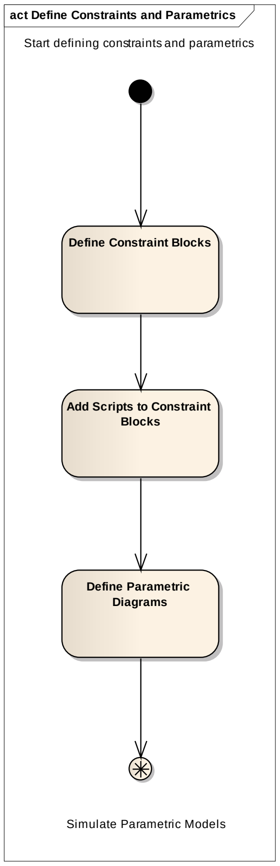

act Define Constraints and Parametrics
Start defining constraints and parametrics
Define Constraint Blocks
Add Scripts to Constraint Blocks
Define Parametric Diagrams
Simulate Parametric Models

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.

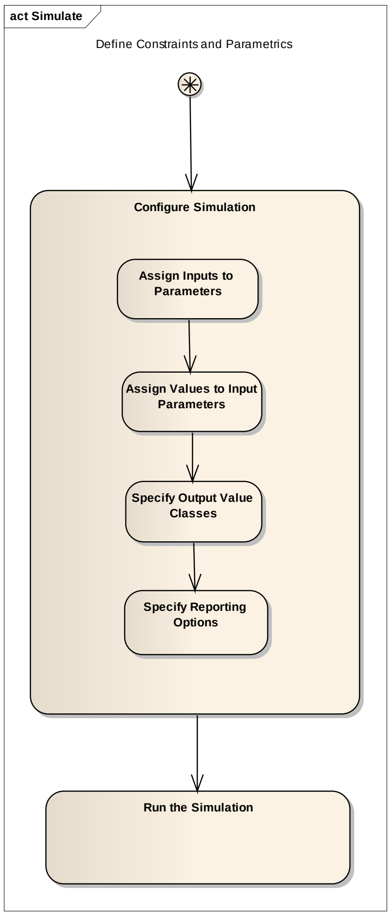
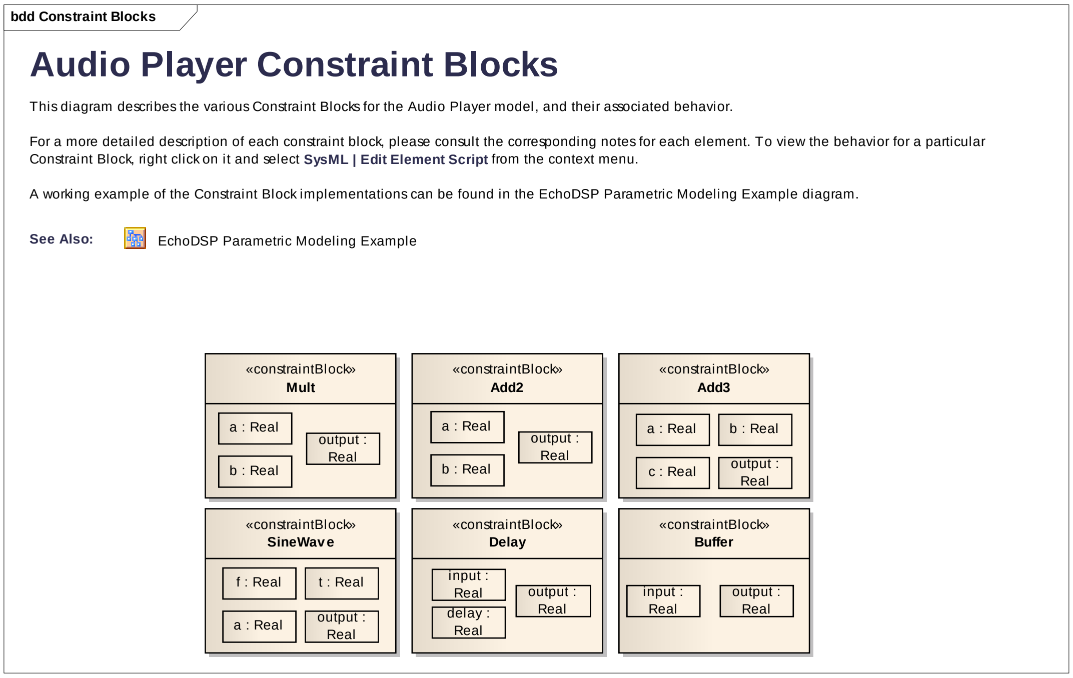

act Simulate
Define Constraints and Parametrics
Configure Simulation
Assign Inputs to Parameters
Assign Values to Input Parameters
Specify Output Value Classes
Specify Reporting Options
Run the Simulation

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.


bdd Constraint Blocks

Audio Player Constraint Blocks

This diagram describes the various Constraint Blocks for the Audio Player model, and their associated behavior.

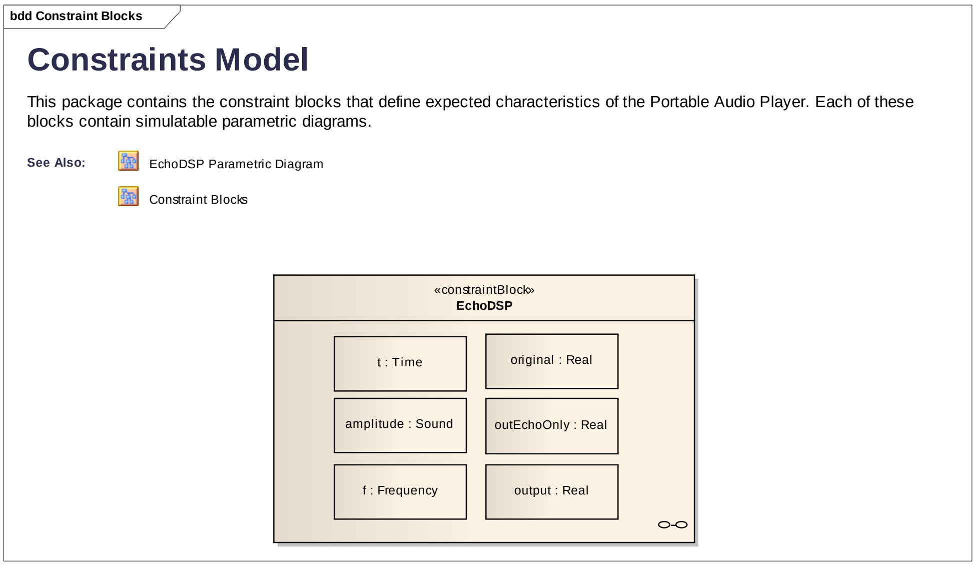
For a more detailed description of each constraint block, please consult the corresponding notes for each element. To view the behavior for a particular Constraint Block, right click on it and select **SysML | Edit Element Script** from the context menu.

A working example of the Constraint Block implementations can be found in the EchoDSP Parametric Modeling Example diagram.

See Also: EchoDSP Parametric Modeling Example

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.


bdd Constraint Blocks

Constraints Model

This package contains the constraint blocks that define expected characteristics of the Portable Audio Player. Each of these blocks contain simulatable parametric diagrams.

See Also: EchoDSP Parametric Diagram
Constraint Blocks

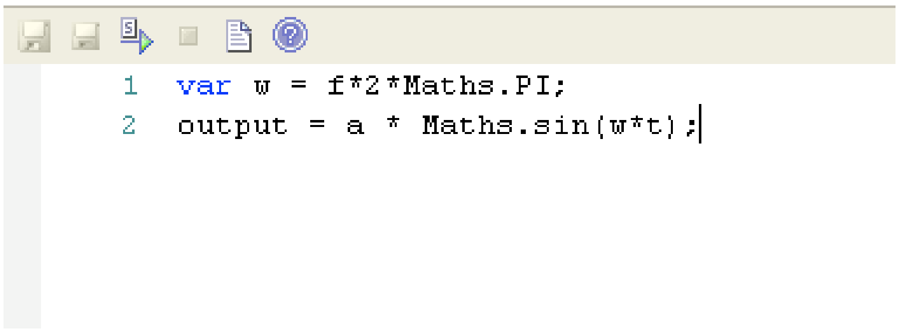
Figure 4 – Constraint Block for the Echo DSP function

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.



```
var w = f*2*Maths.PI;
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.

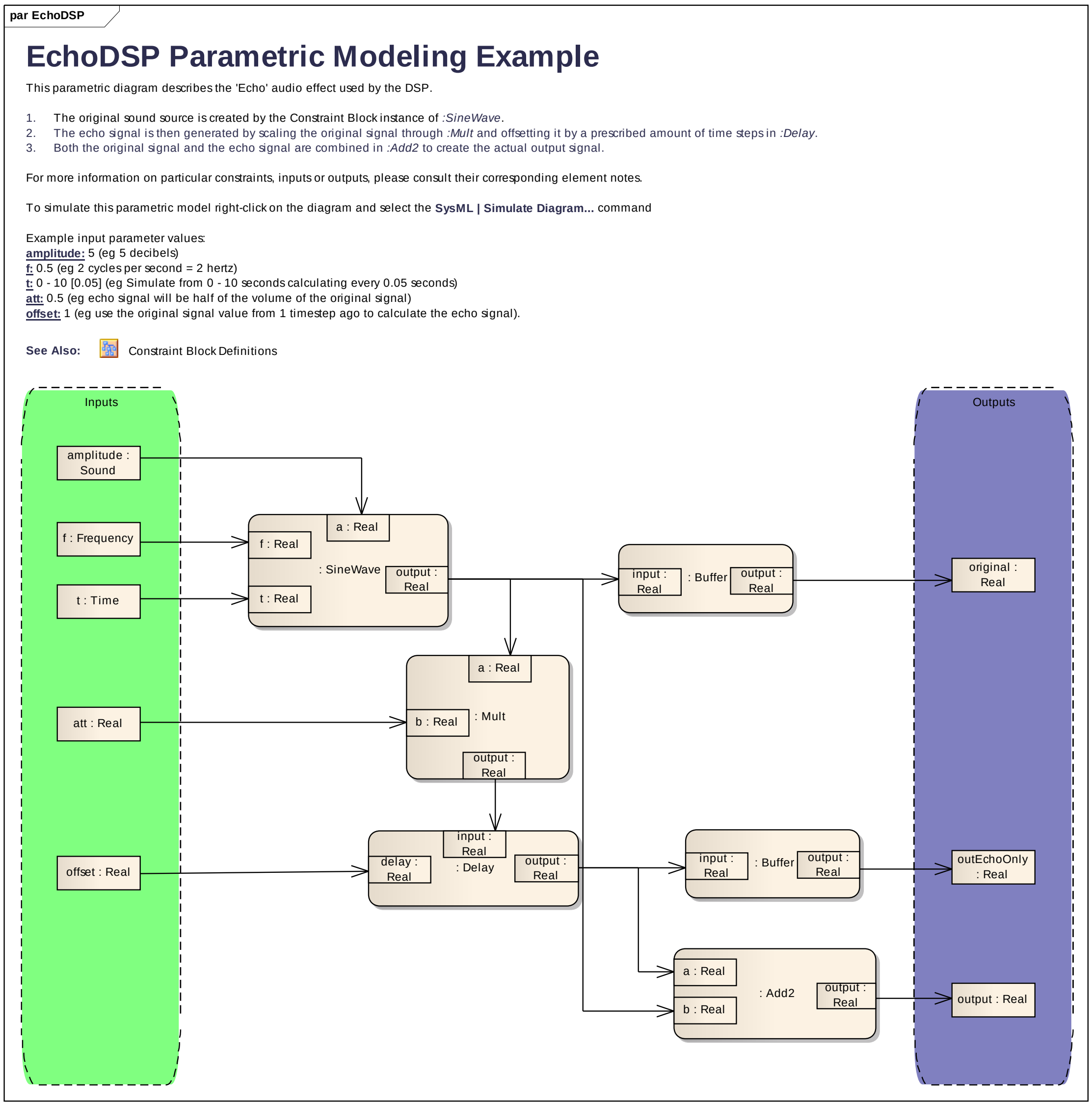
Modeling Tip: Enterprise Architect supports scripting in several languages

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.



par EchoDSP

EchoDSP Parametric Modeling Example

This parametric diagram describes the 'Echo' audio effect used by the DSP.

1. The original sound source is created by the Constraint Block instance of *:SineWave*.
2. The echo signal is then generated by scaling the original signal through *:Mult* and offsetting it by a prescribed amount of time steps in *:Delay*.
3. Both the original signal and the echo signal are combined in *:Add2* to create the actual output signal.

For more information on particular constraints, inputs or outputs, please consult their corresponding element notes.

To simulate this parametric model right-click on the diagram and select the **SysML | Simulate Diagram...** command

Example input parameter values:
amplitude: 5 (eg 5 decibels)
f: 0.5 (eg 2 cycles per second = 2 hertz)
t: 0 - 10 [0.05] (eg Simulate from 0 - 10 seconds calculating every 0.05 seconds)
att: 0.5 (eg echo signal will be half of the volume of the original signal)
offset: 1 (eg use the original signal value from 1 timestep ago to calculate the echo signal).

See Also: Constraint Block Definitions

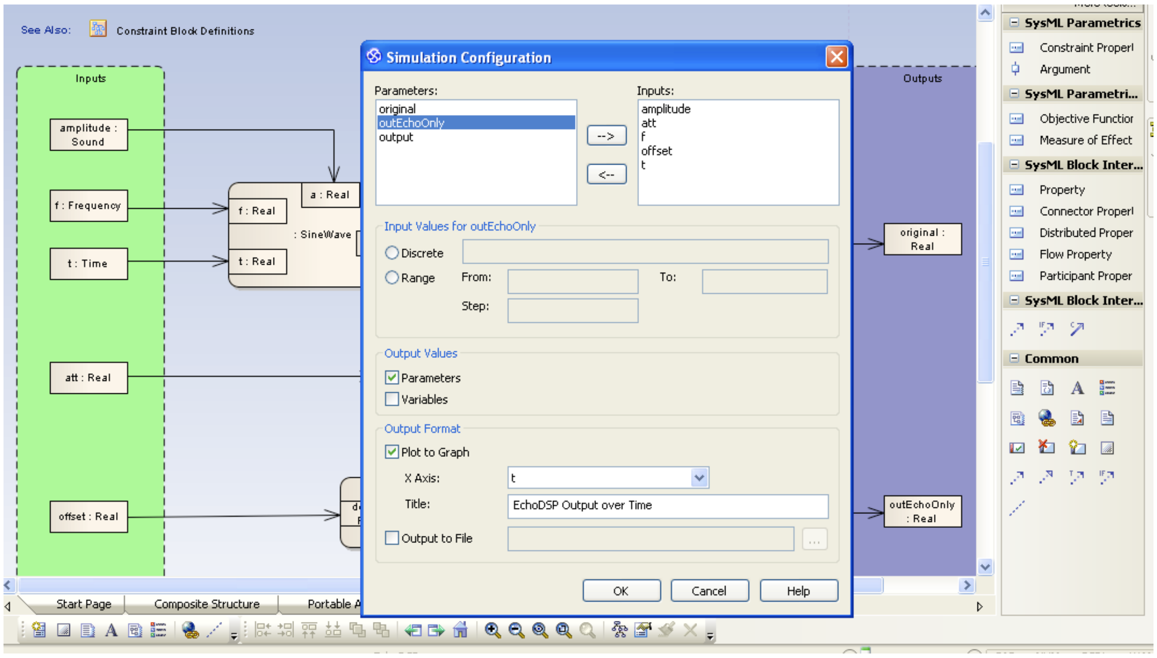
Figure 6 – Parametric Diagram for Echo DSP

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).


Simulation Configuration

Parameters:
original
outEchoOnly
output

Inputs:
amplitude
att
f
offset
t

Input Values for outEchoOnly
Discrete
Range From: To:
Step:

Output Values
[✓] Parameters
[ ] Variables

Output Format
[✓] Plot to Graph
X Axis: t
Title: EchoDSP Output over Time
[ ] Output to File

Figure 7 – Configuring the Simulation

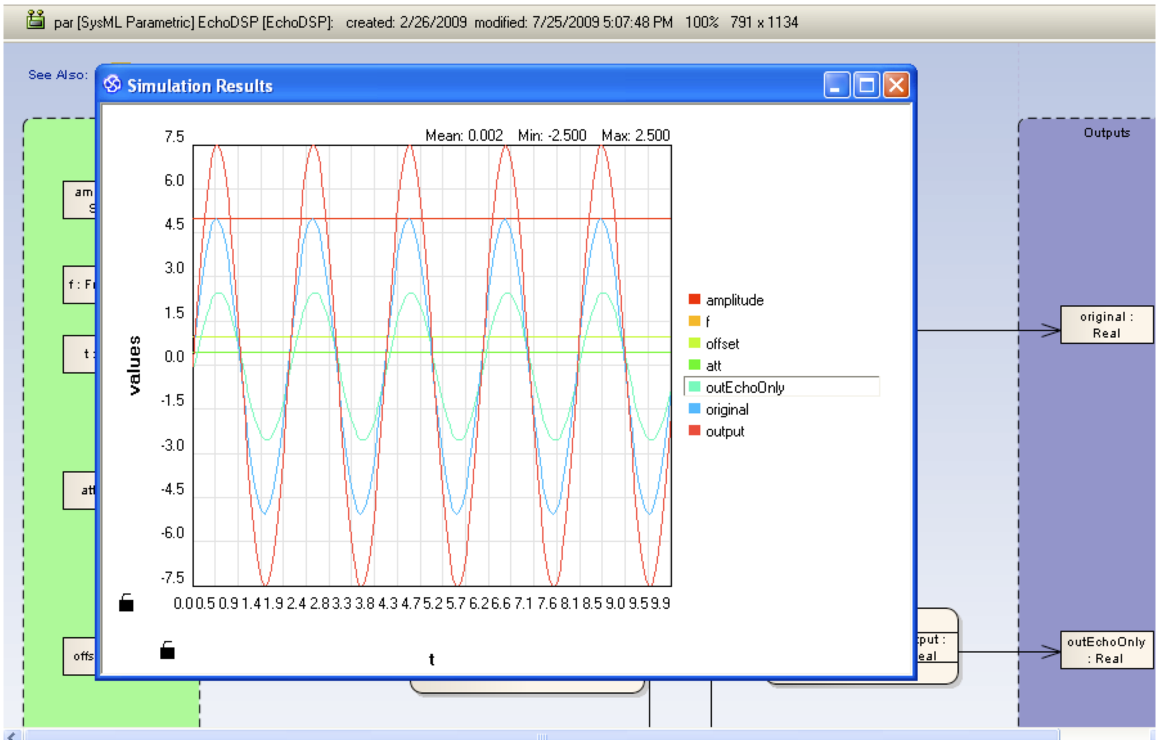
Fill out the Simulation Configuration dialog as follows:

- **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.



par [SysML Parametric] EchoDSP [EchoDSP]: created: 2/26/2009 modified: 7/25/2009 5:07:48 PM 100% 791 x 1134
See Also:
Simulation Results
Mean: 0.002 Min: -2.500 Max: 2.500
values
t
amplitude
f
offset
att
outEchoOnly
original
output
Outputs
original : Real
outEchoOnly : Real

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.

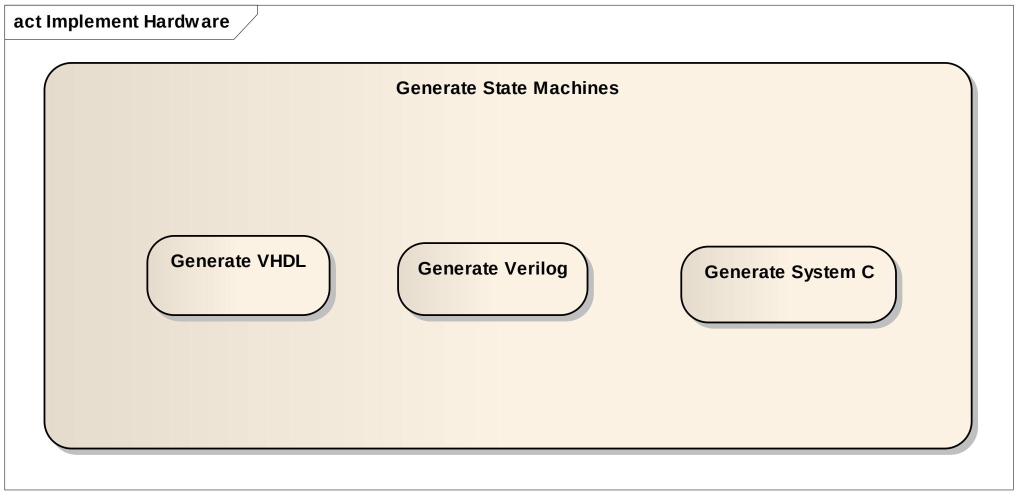
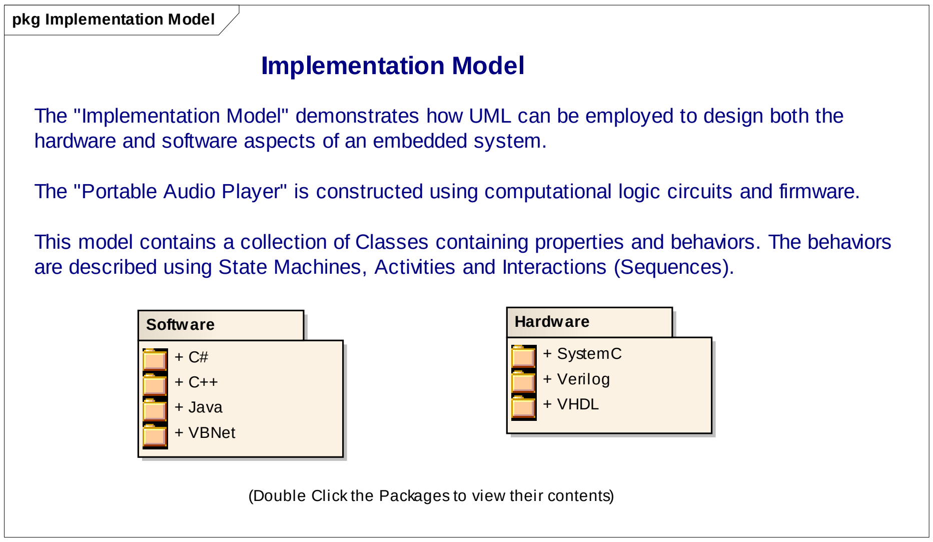


Figure 1 – Hardware Implementation Roadmap with support for three popular Hardware Description Langauges

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.



pkg Implementation Model

Implementation Model

The "Implementation Model" demonstrates how UML can be employed to design both the hardware and software aspects of an embedded system.

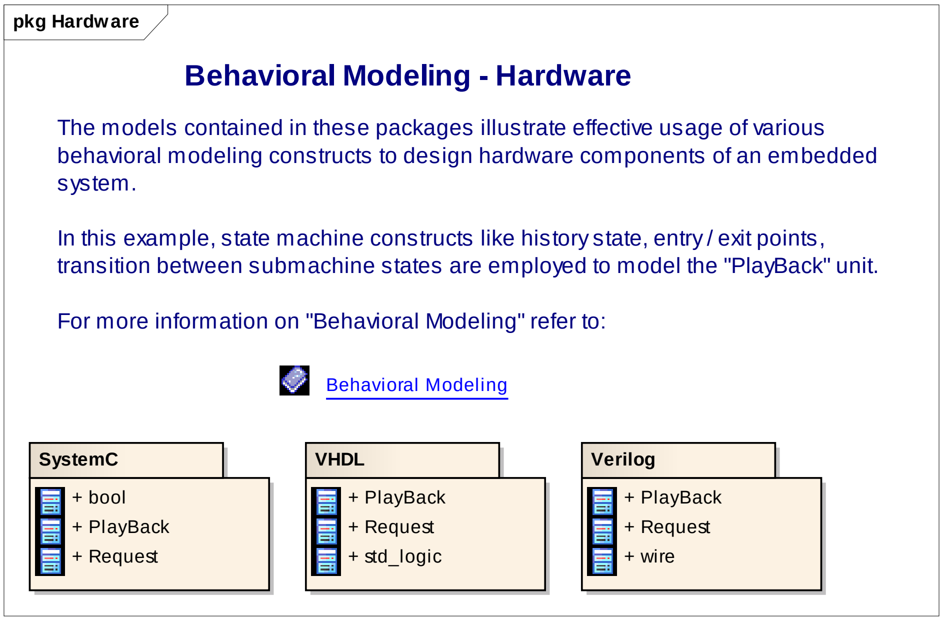
The "Portable Audio Player" is constructed using computational logic circuits and firmware.

This model contains a collection of Classes containing properties and behaviors. The behaviors are described using State Machines, Activities and Interactions (Sequences).

(Double Click the Packages to view their contents)

Figure 2 – Top Level Implementation Package

Let's explore the Hardware package in more detail.



pkg Hardware

Behavioral Modeling - Hardware

The models contained in these packages illustrate effective usage of various behavioral modeling constructs to design hardware components of an embedded system.

In this example, state machine constructs like history state, entry / exit points, transition between submachine states are employed to model the "PlayBack" unit.

For more information on "Behavioral Modeling" refer to:

Behavioral Modeling

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.

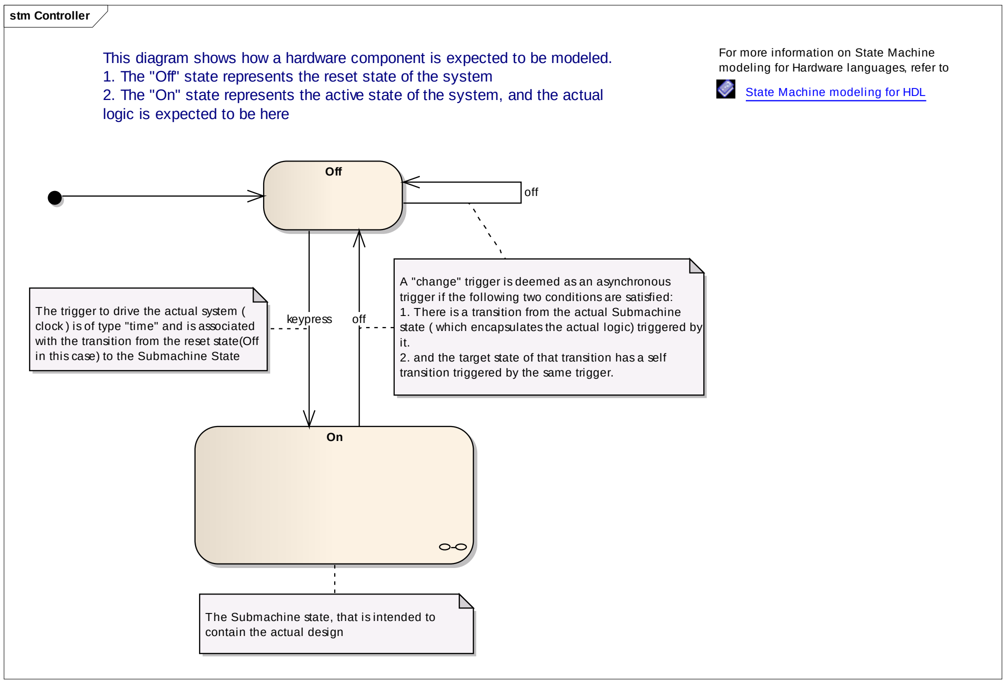

stm Controller
This diagram shows how a hardware component is expected to be modeled.
1. The "Off" state represents the reset state of the system
2. The "On" state represents the active state of the system, and the actual logic is expected to be here
For more information on State Machine modeling for Hardware languages, refer to
State Machine modeling for HDL
Off
off
keypress
off
The trigger to drive the actual system (clock) is of type "time" and is associated with the transition from the reset state(Off in this case) to the Submachine State
A "change" trigger is deemed as an asynchronous trigger if the following two conditions are satisfied:
1. There is a transition from the actual Submachine state (which encapsulates the actual logic) triggered by it.
2. and the target state of that transition has a self transition triggered by the same trigger.
On
The Submachine state, that is intended to contain the actual design

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.

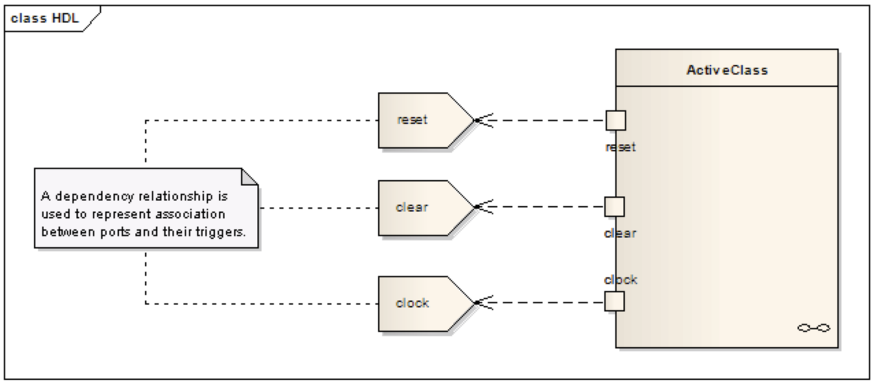

class HDL
ActiveClass
reset
clear
clock
reset
clear
clock
A dependency relationship is used to represent association between ports and their triggers.

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.

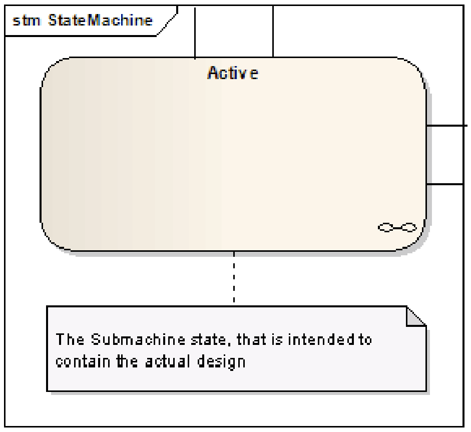

stm StateMachine
Active
The Submachine state, that is intended to contain the actual design

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.

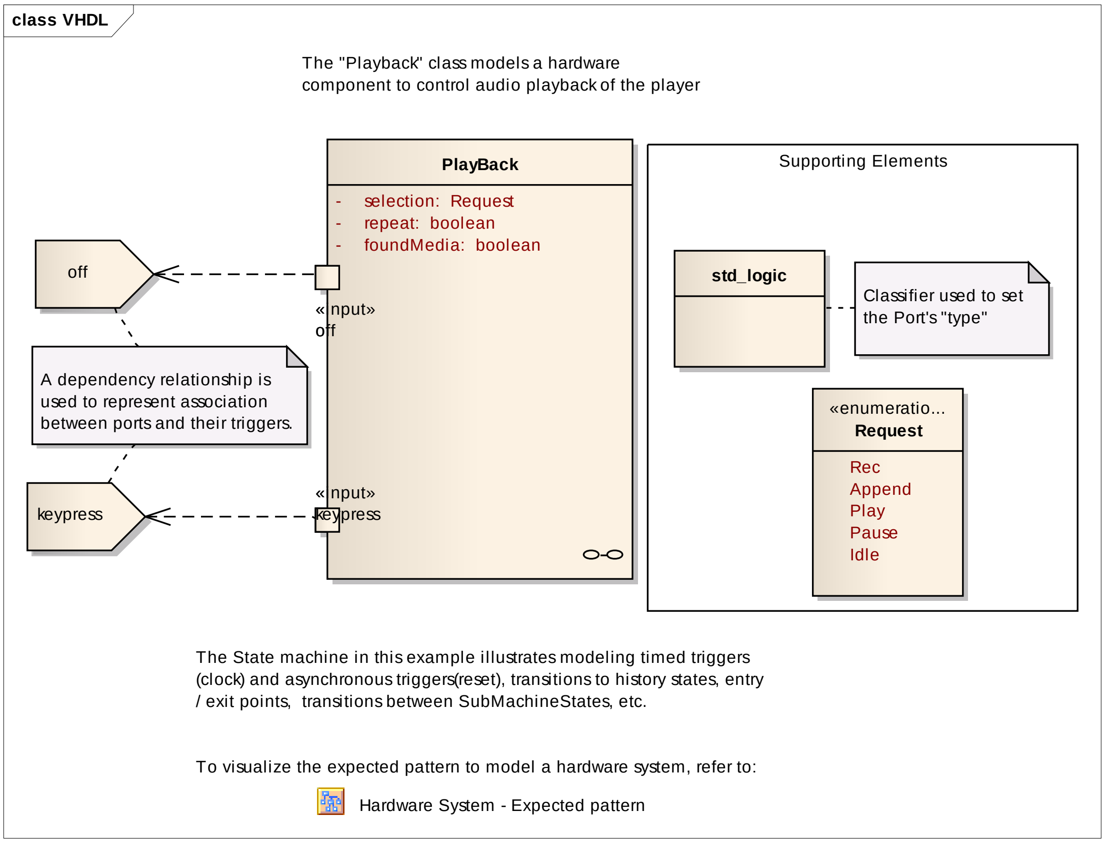

class VHDL
The "Playback" class models a hardware component to control audio playback of the player
PlayBack
- selection: Request
- repeat: boolean
- foundMedia: boolean
off
«input» off
A dependency relationship is used to represent association between ports and their triggers.
keypress
«input» keypress
Supporting Elements
std_logic
Classifier used to set the Port's "type"
«enumeratio... Request
Rec
Append
Play
Pause
Idle
The State machine in this example illustrates modeling timed triggers (clock) and asynchronous triggers(reset), transitions to history states, entry / exit points, transitions between SubMachineStates, etc.
To visualize the expected pattern to model a hardware system, refer to:
Hardware System - Expected pattern

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

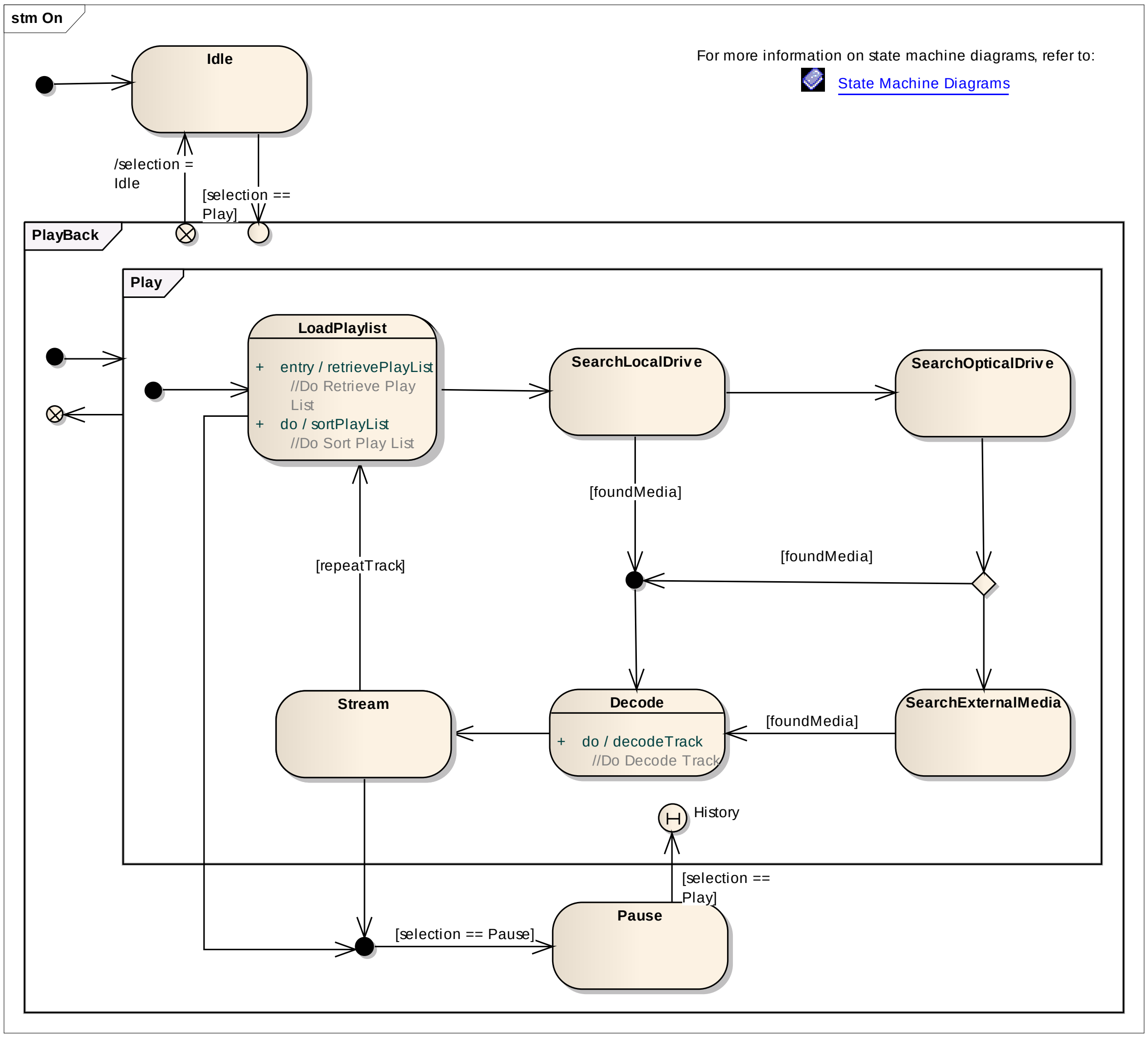


Figure 9 – State Machine for Playback

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.

An Overview of VHDL

Here's a brief summary of VHDL that we extracted from Wikipedia.[6]

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.

[6] 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
architecture	Class	An architecture
asynchronous	Method	An asynchronous process
configuration	Method	A configuration
enumeration	Inner Class	An *enum* type
entity	Interface	An entity
part	Attribute	A component instantiation
port	Attribute	A port
signal	Attribute	A signal declaration
struct	Inner Class	A record definition
synchronous	Method	A synchronous process
typedef	Inner Class	A *type* or *subtype* definition

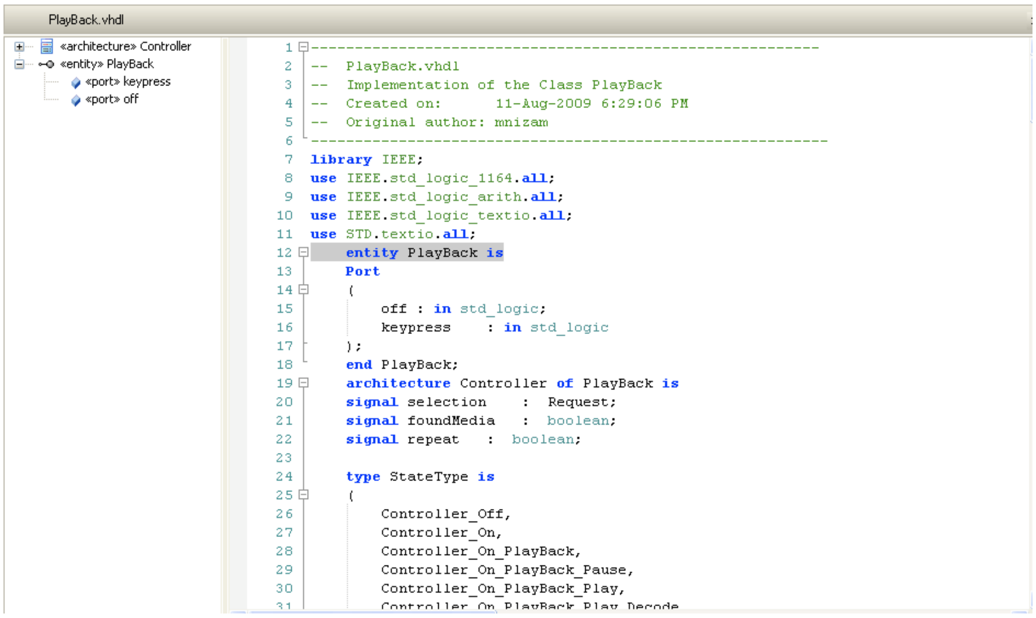
Figure 10 – VHDL Stereotypes used by Enterprise Architect

Tagged Values

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

Figure 11 – VHDL Tagged Values used by Enterprise Architect

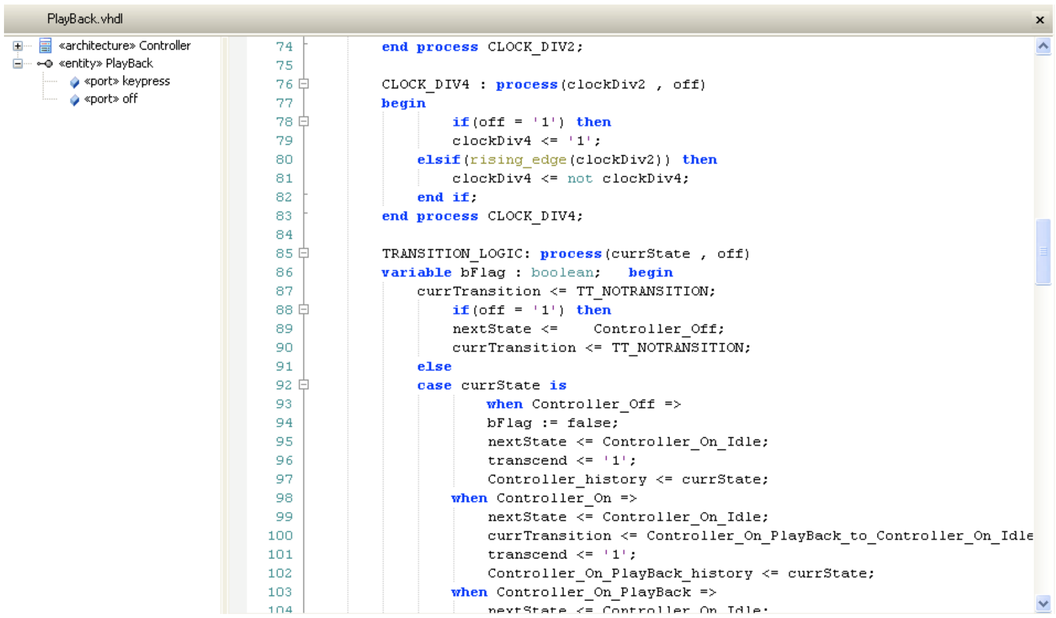
Figures 12 and 13 show a portion of the generated code produced by Enterprise Architect.



```
----------------------------------------------------------
--  PlayBack.vhdl
--  Implementation of the Class PlayBack
--  Created on:      11-Aug-2009 6:29:06 PM
--  Original author: mnizam
-----------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;
use STD.textio.all;
    entity PlayBack is
    Port
    (
        off : in std_logic;
        keypress    : in std_logic
    );
    end PlayBack;
    architecture Controller of PlayBack is
    signal selection    :  Request;
    signal foundMedia   :  boolean;
    signal repeat   :  boolean;

    type StateType is
    (
        Controller_Off,
        Controller_On,
        Controller_On_PlayBack,
        Controller_On_PlayBack_Pause,
        Controller_On_PlayBack_Play,
        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.



```
        end process CLOCK_DIV2;

        CLOCK_DIV4 : process(clockDiv2 , off)
        begin
                if(off = '1') then
                clockDiv4 <= '1';
            elsif(rising_edge(clockDiv2)) then
                clockDiv4 <= not clockDiv4;
            end if;
        end process CLOCK_DIV4;

        TRANSITION_LOGIC: process(currState , off)
        variable bFlag : boolean;   begin
            currTransition <= TT_NOTRANSITION;
                if(off = '1') then
                nextState <=    Controller_Off;
                currTransition <= TT_NOTRANSITION;
            else
            case currState is
                    when Controller_Off =>
                    bFlag := false;
                    nextState <= Controller_On_Idle;
                    transcend <= '1';
                    Controller_history <= currState;
                when Controller_On =>
                    nextState <= Controller_On_Idle;
                    currTransition <= Controller_On_PlayBack_to_Controller_On_Idle
                    transcend <= '1';
                    Controller_On_PlayBack_history <= currState;
                when Controller_On_PlayBack =>
                    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

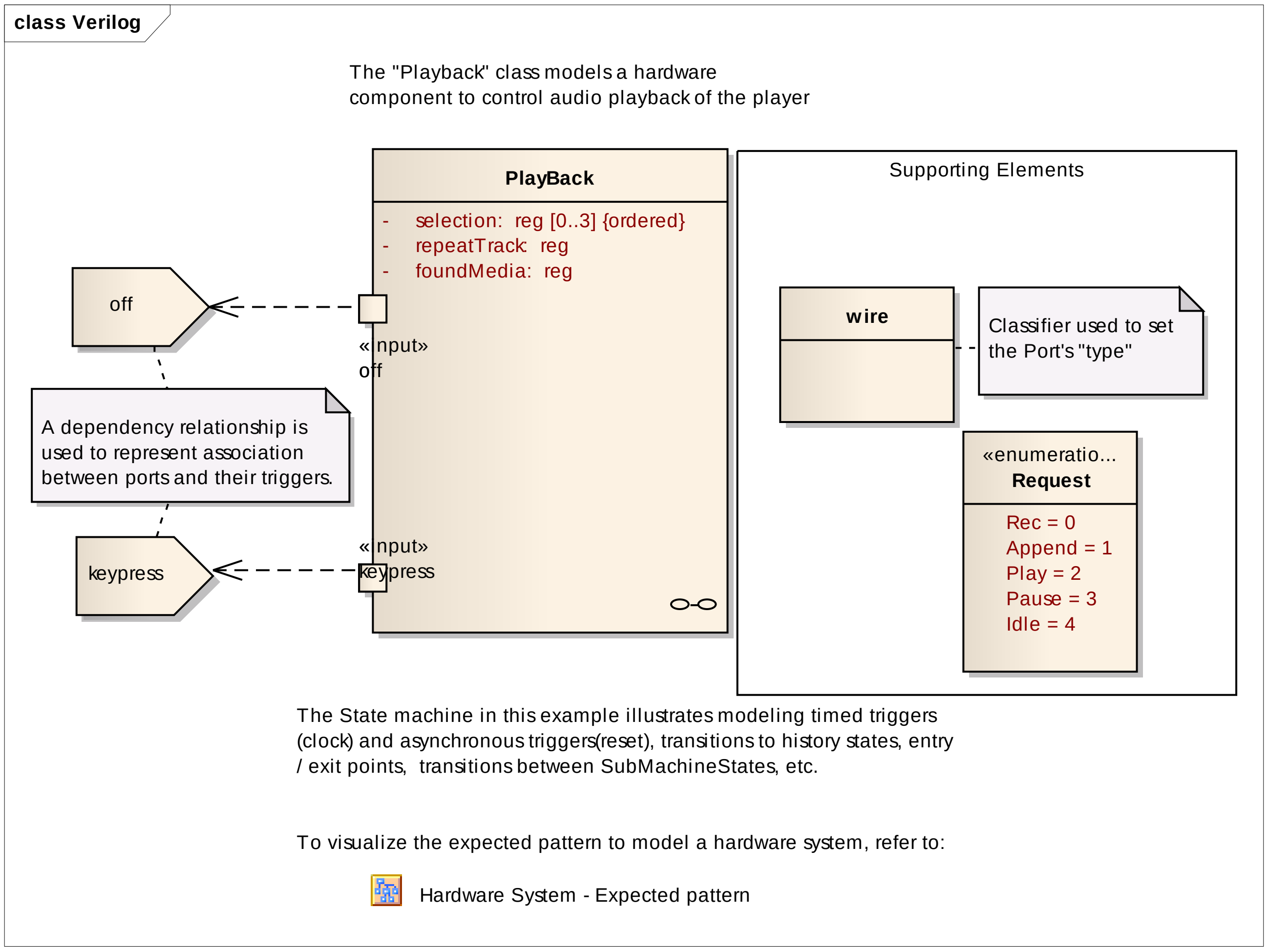

class Verilog
The "Playback" class models a hardware component to control audio playback of the player
PlayBack
- selection: reg [0..3] {ordered}
- repeatTrack: reg
- foundMedia: reg
off
«input» off
A dependency relationship is used to represent association between ports and their triggers.
keypress
«input» keypress
Supporting Elements
wire
Classifier used to set the Port's "type"
«enumeratio...
Request
Rec = 0
Append = 1
Play = 2
Pause = 3
Idle = 4
The State machine in this example illustrates modeling timed triggers (clock) and asynchronous triggers(reset), transitions to history states, entry / exit points, transitions between SubMachineStates, etc.
To visualize the expected pattern to model a hardware system, refer to:
Hardware System - Expected pattern

Figure 14 – Playback class diagram for Verilog implementation

An Overview of Verilog

We consulted Wikipedia again[7] *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 descr iption 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

[7] 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 synthsizeable 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

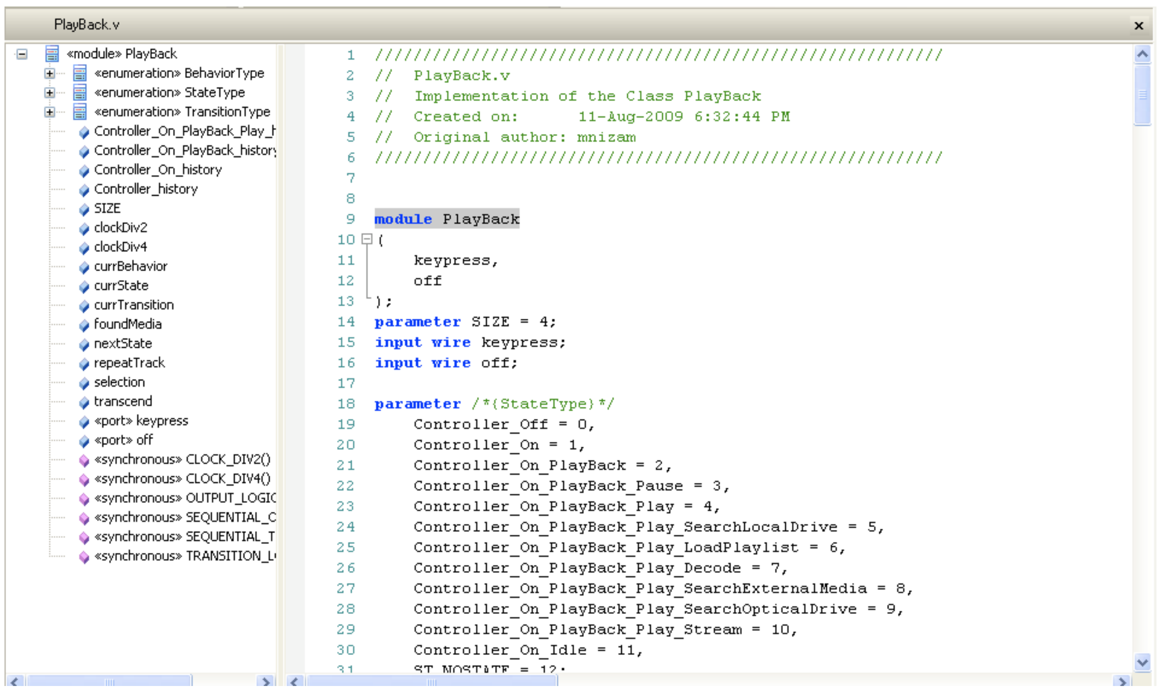
Stereotype	Applies To	Corresponds To
asynchronous	Method	A concurrent process
enumeration	Inner Class	An *enum* type
initializer	Method	An initializer process
module	Class	A module
part	Attribute	A component instantiation
port	Attribute	A port
synchronous	Method	A sequential process

Figure 15 – Verilog Stereotypes used by Enterprise Architect

Tagged Values

Tag	Applies To	Corresponds To
kind	Attribute (signal)	The signal kind (such as *register*, *bus*)
mode	Attribute (port)	The port mode (*in*, *out*, *inout*)
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



PlayBack.v

```
«module» PlayBack
    «enumeration» BehaviorType
    «enumeration» StateType
    «enumeration» TransitionType
    Controller_On_PlayBack_Play_
    Controller_On_PlayBack_histor
    Controller_On_history
    Controller_history
    SIZE
    clockDiv2
    clockDiv4
    currBehavior
    currState
    currTransition
    foundMedia
    nextState
    repeatTrack
    selection
    transcend
    «port» keypress
    «port» off
    «synchronous» CLOCK_DIV2()
    «synchronous» CLOCK_DIV4()
    «synchronous» OUTPUT_LOGIC
    «synchronous» SEQUENTIAL_C
    «synchronous» SEQUENTIAL_T
    «synchronous» TRANSITION_L
```

```
///////////////////////////////////////////////////////////
//  PlayBack.v
//  Implementation of the Class PlayBack
//  Created on:      11-Aug-2009 6:32:44 PM
//  Original author: mnizam
///////////////////////////////////////////////////////////


module PlayBack
(
    keypress,
    off
);
parameter SIZE = 4;
input wire keypress;
input wire off;

parameter /*{StateType}*/
    Controller_Off = 0,
    Controller_On = 1,
    Controller_On_PlayBack = 2,
    Controller_On_PlayBack_Pause = 3,
    Controller_On_PlayBack_Play = 4,
    Controller_On_PlayBack_Play_SearchLocalDrive = 5,
    Controller_On_PlayBack_Play_LoadPlaylist = 6,
    Controller_On_PlayBack_Play_Decode = 7,
    Controller_On_PlayBack_Play_SearchExternalMedia = 8,
    Controller_On_PlayBack_Play_SearchOpticalDrive = 9,
    Controller_On_PlayBack_Play_Stream = 10,
    Controller_On_Idle = 11,
    ST_NOSTATE = 12;
```

Figure 17 – Verilog code generated by Enterprise Architect

Implementation in SystemC

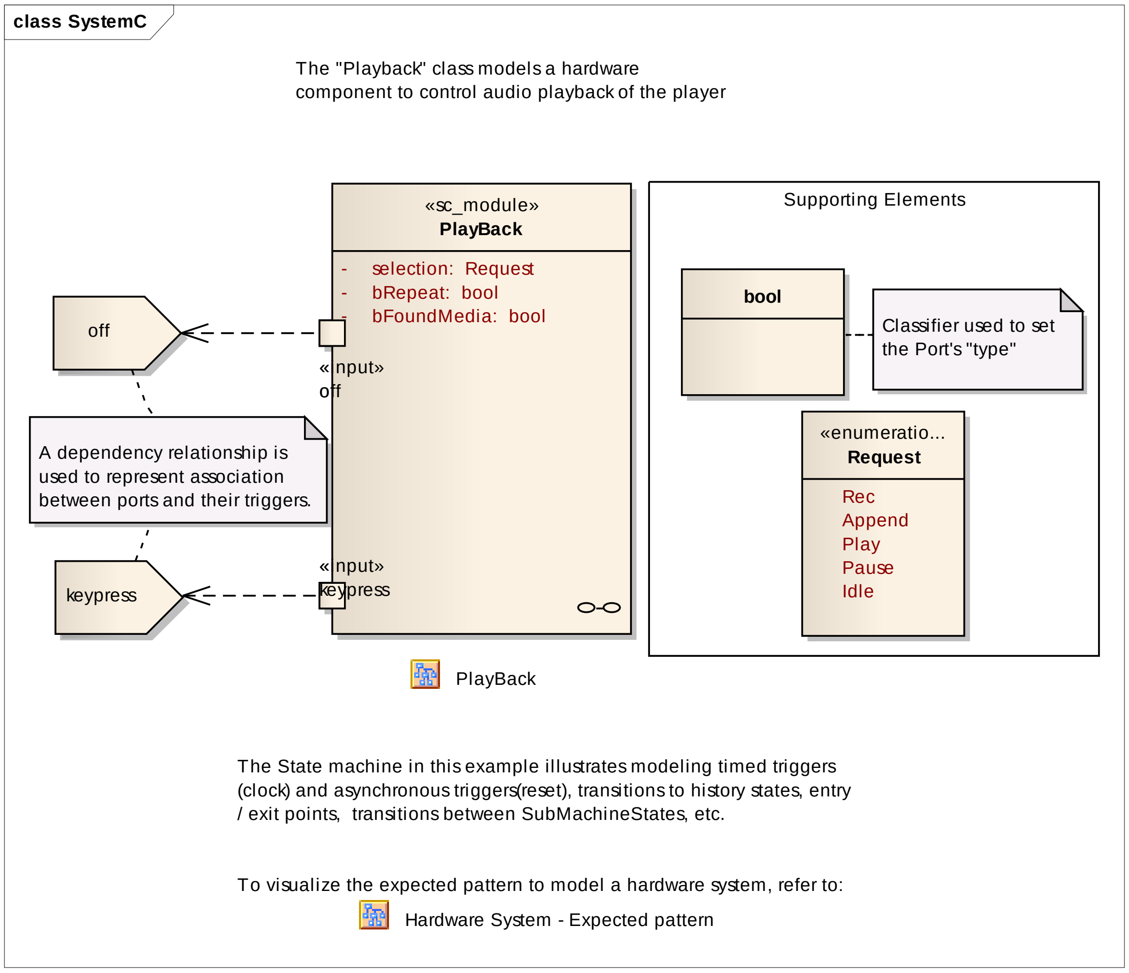

class SystemC
The "Playback" class models a hardware component to control audio playback of the player
«sc_module»
PlayBack
- selection: Request
- bRepeat: bool
- bFoundMedia: bool
«input»
off
«input»
keypress
off
keypress
A dependency relationship is used to represent association between ports and their triggers.
Supporting Elements
bool
Classifier used to set the Port's "type"
«enumeratio...
Request
Rec
Append
Play
Pause
Idle
PlayBack
The State machine in this example illustrates modeling timed triggers (clock) and asynchronous triggers(reset), transitions to history states, entry / exit points, transitions between SubMachineStates, etc.
To visualize the expected pattern to model a hardware system, refer to:
Hardware System - Expected pattern

Figure 18 – Playback class diagram for SystemC implementation

An Overview of SystemC

We've consulted Wikipedia one final time[8] 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).

[8] 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[9].

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
delegate	Method	A delegate.
enumeration	Inner Class	An *enum* type.
friend	Method	A *friend* method.
property	Method	A property definition.
sc_ctor	Method	A SystemC constructor.
sc_module	Class	A SystemC module.
sc_port	Attribute	A port.
sc_signal	Attribute	A signal
struct	Inner Class	A *struct* or *union*.

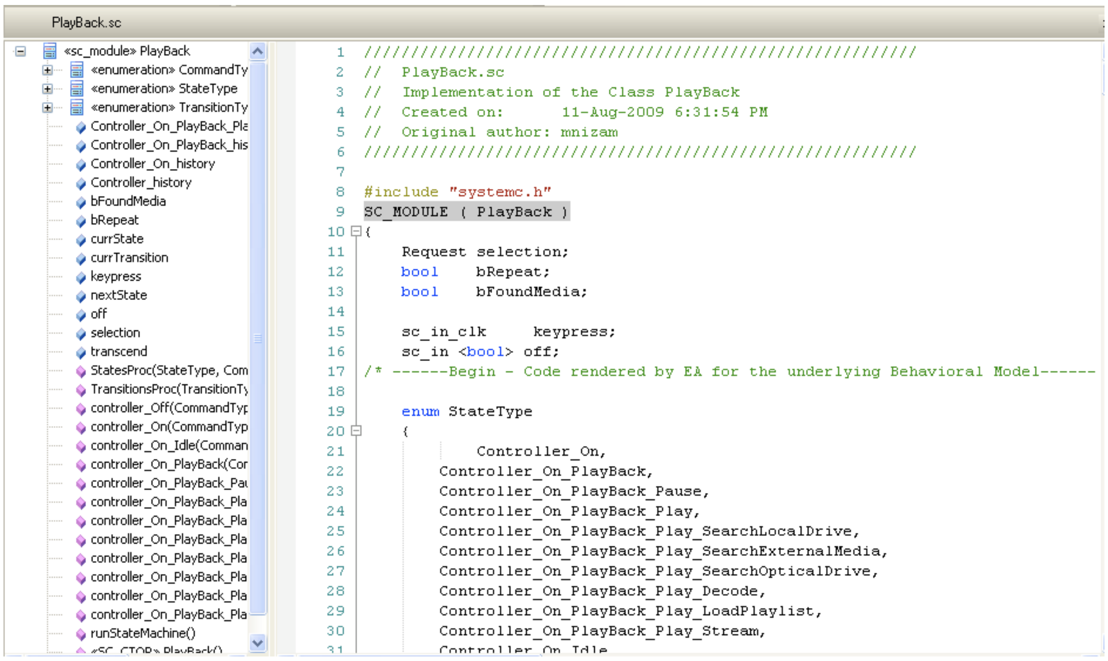
Figure 19 – SystemC stereotypes used by Enterprise Architect

[9] http://www.systemc.org/home/

Tagged Values

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

Figure 20 – SystemC tagged values used by Enterprise Architect



PlayBack.sc

```
«sc_module» PlayBack
  «enumeration» CommandTy
  «enumeration» StateType
  «enumeration» TransitionTy
  Controller_On_PlayBack_Pla
  Controller_On_PlayBack_his
  Controller_On_history
  Controller_history
  bFoundMedia
  bRepeat
  currState
  currTransition
  keypress
  nextState
  off
  selection
  transcend
  StatesProc(StateType, Com
  TransitionsProc(TransitionTy
  controller_Off(CommandTyp
  controller_On(CommandTyp
  controller_On_Idle(Comman
  controller_On_PlayBack(Cor
  controller_On_PlayBack_Pau
  controller_On_PlayBack_Pla
  controller_On_PlayBack_Pla
  controller_On_PlayBack_Pla
  controller_On_PlayBack_Pla
  controller_On_PlayBack_Pla
  controller_On_PlayBack_Pla
  controller_On_PlayBack_Pla
  runStateMachine()
```

```
////////////////////////////////////////////////////////////
//  PlayBack.sc
//  Implementation of the Class PlayBack
//  Created on:      11-Aug-2009 6:31:54 PM
//  Original author: mnizam
////////////////////////////////////////////////////////////

#include "systemc.h"
SC_MODULE ( PlayBack )
{
    Request selection;
    bool    bRepeat;
    bool    bFoundMedia;

    sc_in_clk     keypress;
    sc_in <bool> off;
/* ------Begin - Code rendered by EA for the underlying Behavioral Model------

    enum StateType
    {
            Controller_On,
        Controller_On_PlayBack,
        Controller_On_PlayBack_Pause,
        Controller_On_PlayBack_Play,
        Controller_On_PlayBack_Play_SearchLocalDrive,
        Controller_On_PlayBack_Play_SearchExternalMedia,
        Controller_On_PlayBack_Play_SearchOpticalDrive,
        Controller_On_PlayBack_Play_Decode,
        Controller_On_PlayBack_Play_LoadPlaylist,
        Controller_On_PlayBack_Play_Stream,
        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.[10] 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

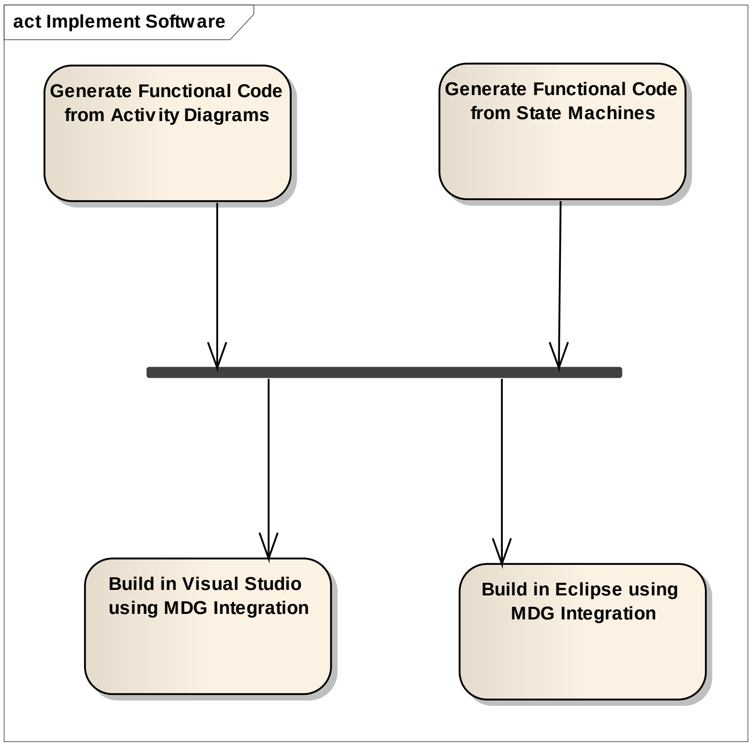

act Implement Software
Generate Functional Code from Activity Diagrams
Generate Functional Code from State Machines
Build in Visual Studio using MDG Integration
Build in Eclipse using MDG Integration

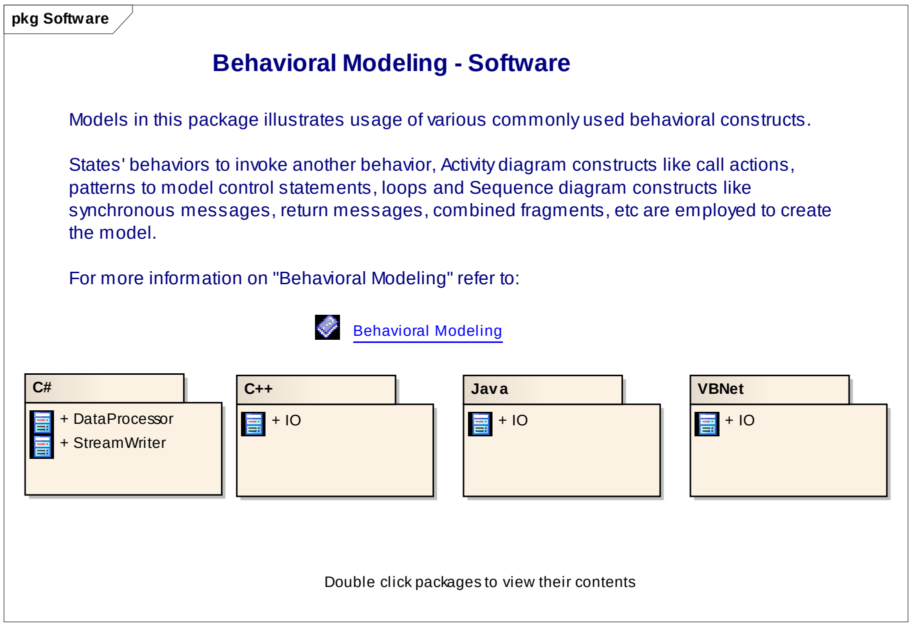
Figure 1 – Roadmap for Software Implementation

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.

[10] *Enterprise Architect for Power Users* multimedia tutorial: www.iconixsw.com/EA/PowerUsers.html



pkg Software
Behavioral Modeling - Software
Models in this package illustrates usage of various commonly used behavioral constructs.
States' behaviors to invoke another behavior, Activity diagram constructs like call actions, patterns to model control statements, loops and Sequence diagram constructs like synchronous messages, return messages, combined fragments, etc are employed to create the model.
For more information on "Behavioral Modeling" refer to:
Behavioral Modeling
C#
+ DataProcessor
+ StreamWriter
C++
+ IO
Java
+ IO
VBNet
+ IO
Double click packages to view their contents

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

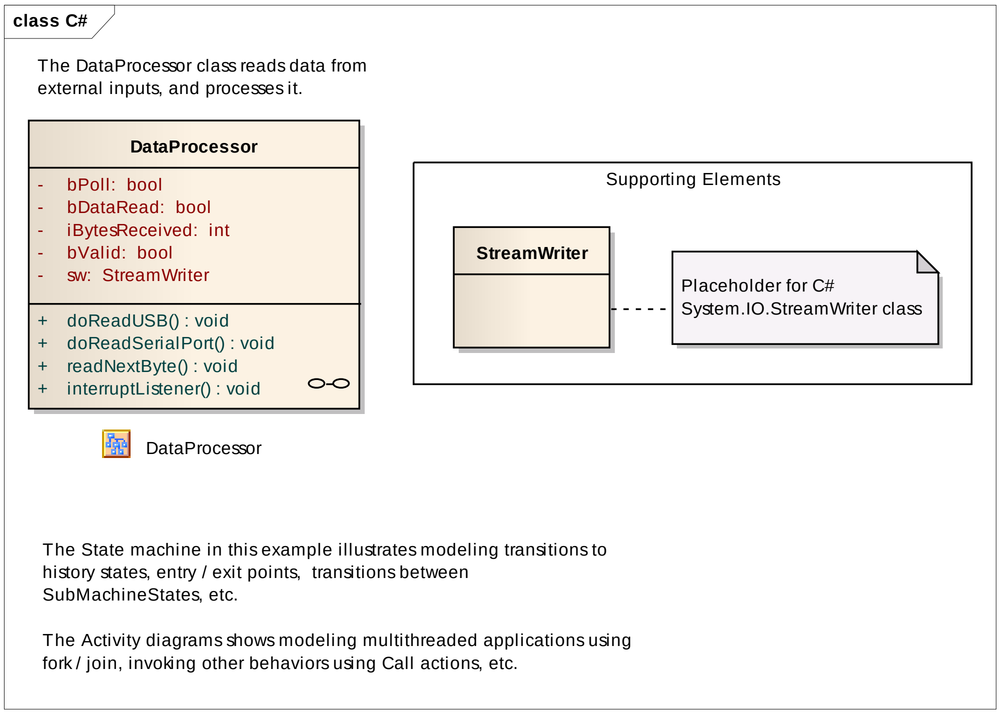

class C#
The DataProcessor class reads data from external inputs, and processes it.
DataProcessor
- bPoll: bool
- bDataRead: bool
- iBytesReceived: int
- bValid: bool
- sw: StreamWriter
+ doReadUSB() : void
+ doReadSerialPort() : void
+ readNextByte() : void
+ interruptListener() : void
Supporting Elements
StreamWriter
Placeholder for C# System.IO.StreamWriter class
DataProcessor
The State machine in this example illustrates modeling transitions to history states, entry / exit points, transitions between SubMachineStates, etc.
The Activity diagrams shows modeling multithreaded applications using fork / join, invoking other behaviors using Call actions, etc.

Figure 3 – DataProcessor Class Diagram

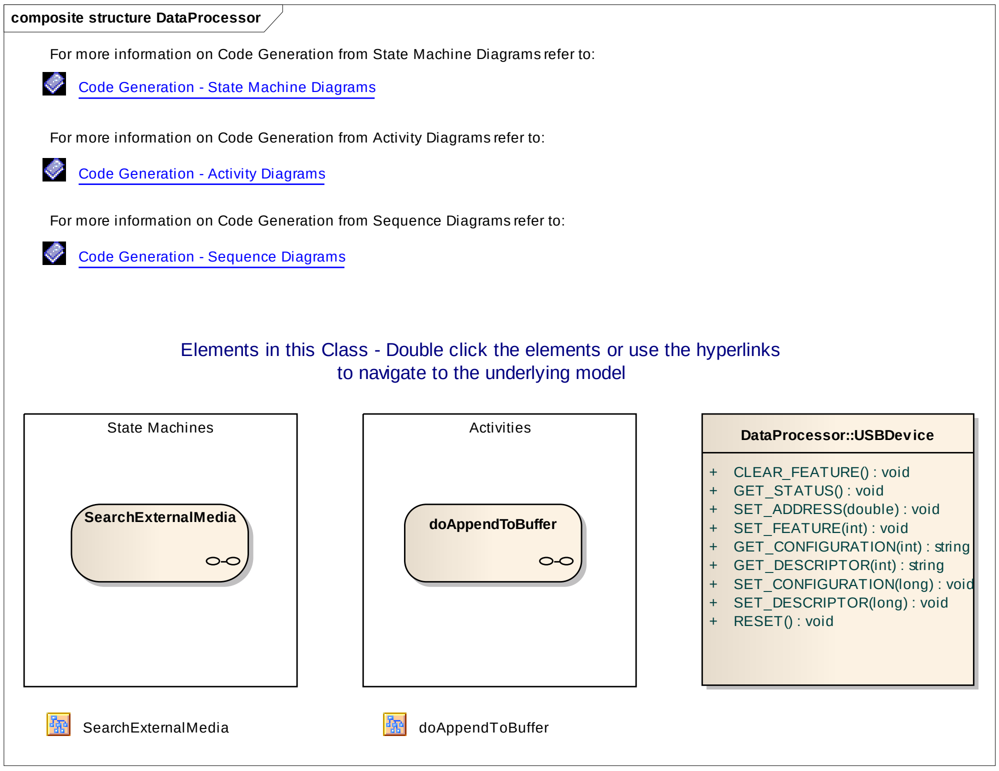

composite structure DataProcessor
For more information on Code Generation from State Machine Diagrams refer to:
Code Generation - State Machine Diagrams
For more information on Code Generation from Activity Diagrams refer to:
Code Generation - Activity Diagrams
For more information on Code Generation from Sequence Diagrams refer to:
Code Generation - Sequence Diagrams
Elements in this Class - Double click the elements or use the hyperlinks to navigate to the underlying model
State Machines
SearchExternalMedia
Activities
doAppendToBuffer
DataProcessor::USBDevice
+ CLEAR_FEATURE() : void
+ GET_STATUS() : void
+ SET_ADDRESS(double) : void
+ SET_FEATURE(int) : void
+ GET_CONFIGURATION(int) : string
+ GET_DESCRIPTOR(int) : string
+ SET_CONFIGURATION(long) : void
+ SET_DESCRIPTOR(long) : void
+ RESET() : void
SearchExternalMedia
doAppendToBuffer

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.

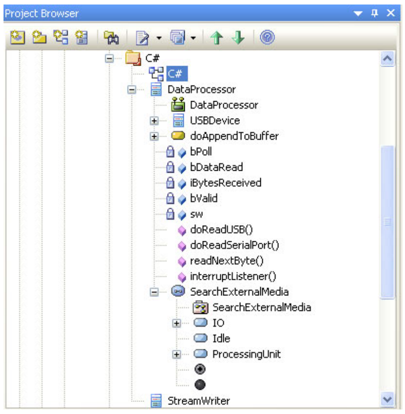


Figure 5 – Behaviors to be code generated are nested within a parent class

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.

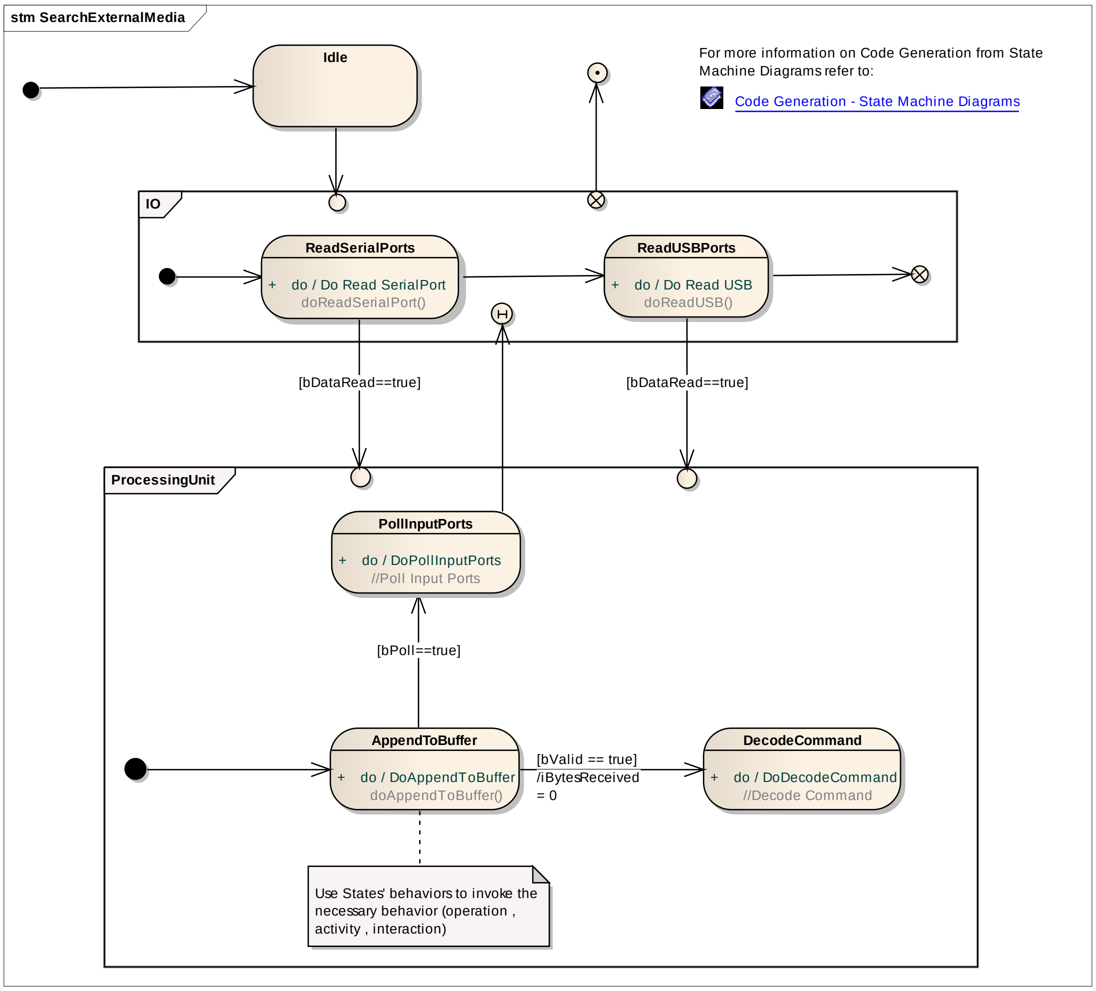
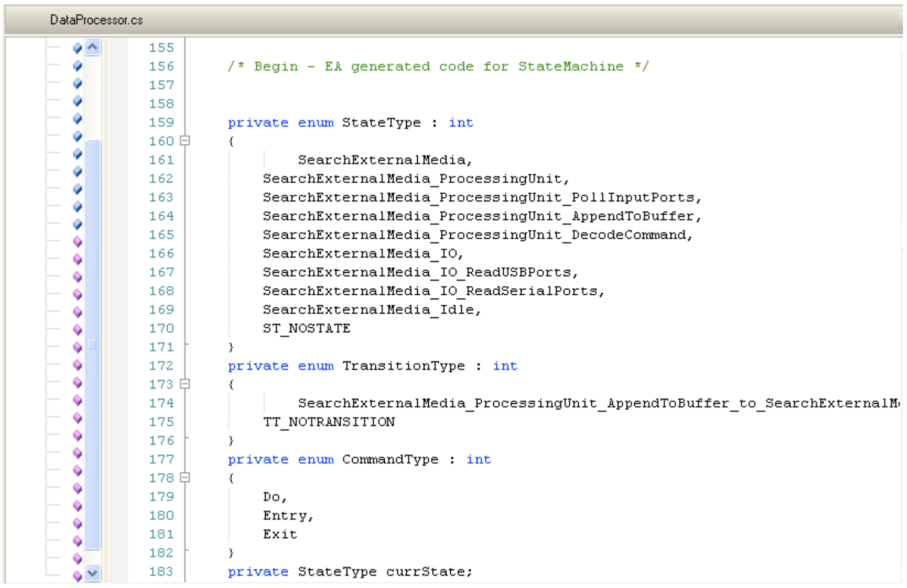


Figure 6 – State Machine for Searching External Media

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.



DataProcessor.cs

```
/* Begin - EA generated code for StateMachine */


private enum StateType : int
{
        SearchExternalMedia,
    SearchExternalMedia_ProcessingUnit,
    SearchExternalMedia_ProcessingUnit_PollInputPorts,
    SearchExternalMedia_ProcessingUnit_AppendToBuffer,
    SearchExternalMedia_ProcessingUnit_DecodeCommand,
    SearchExternalMedia_IO,
    SearchExternalMedia_IO_ReadUSBPorts,
    SearchExternalMedia_IO_ReadSerialPorts,
    SearchExternalMedia_Idle,
    ST_NOSTATE
}
private enum TransitionType : int
{
        SearchExternalMedia_ProcessingUnit_AppendToBuffer_to_SearchExternalM
    TT_NOTRANSITION
}
private enum CommandType : int
{
    Do,
    Entry,
    Exit
}
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.

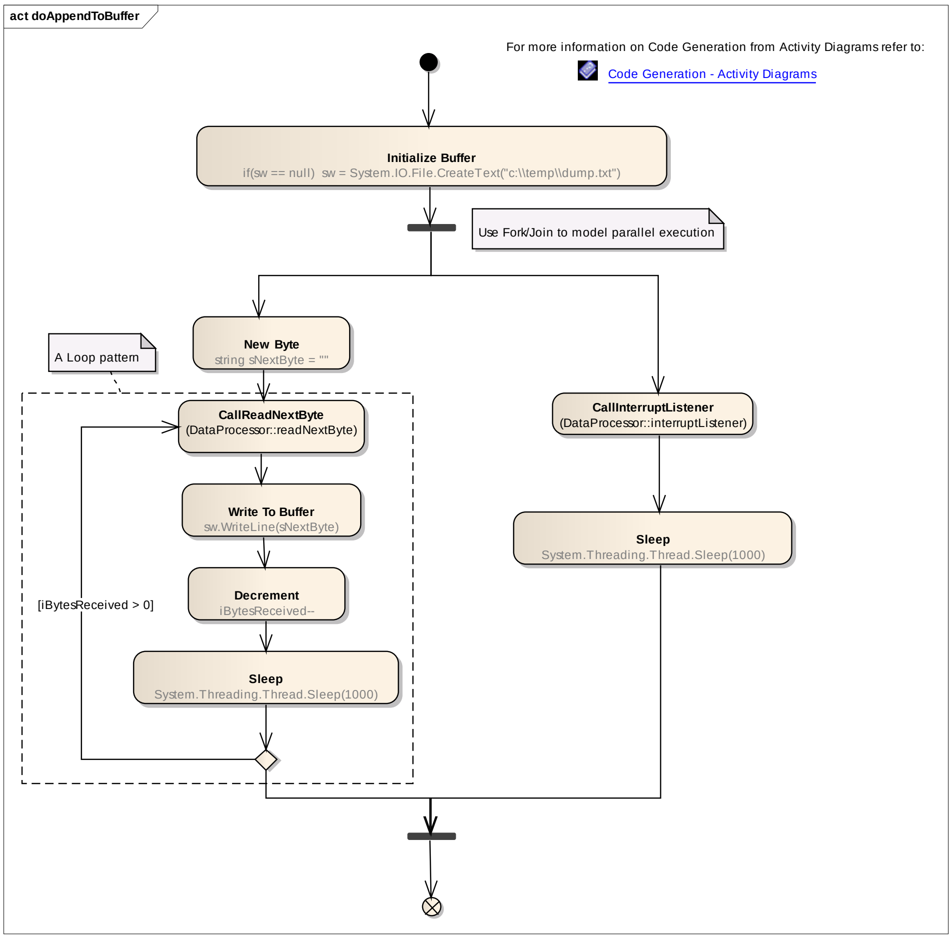
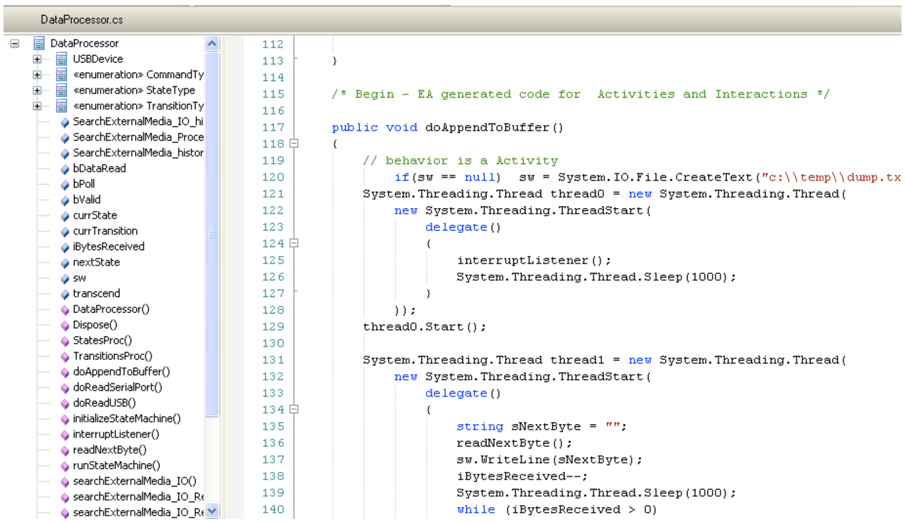

act doAppendToBuffer
For more information on Code Generation from Activity Diagrams refer to:
Code Generation - Activity Diagrams
Initialize Buffer
if(sw == null) sw = System.IO.File.CreateText("c:\\temp\\dump.txt")
Use Fork/Join to model parallel execution
A Loop pattern
New Byte
string sNextByte = ""
CallReadNextByte
(DataProcessor::readNextByte)
Write To Buffer
sw.WriteLine(sNextByte)
Decrement
iBytesReceived--
Sleep
System.Threading.Thread.Sleep(1000)
[iBytesReceived > 0]
CallInterruptListener
(DataProcessor::interruptListener)
Sleep
System.Threading.Thread.Sleep(1000)

Figure 8 – Activity Diagram for AppendToBuffer



DataProcessor.cs

```
DataProcessor
    USBDevice
    «enumeration» CommandTy
    «enumeration» StateType
    «enumeration» TransitionTy
    SearchExternalMedia_IO_hi
    SearchExternalMedia_Proce
    SearchExternalMedia_histor
    bDataRead
    bPoll
    bValid
    currState
    currTransition
    iBytesReceived
    nextState
    sw
    transcend
    DataProcessor()
    Dispose()
    StatesProc()
    TransitionsProc()
    doAppendToBuffer()
    doReadSerialPort()
    doReadUSB()
    initializeStateMachine()
    interruptListener()
    readNextByte()
    runStateMachine()
    searchExternalMedia_IO()
    searchExternalMedia_IO_Re
    searchExternalMedia_IO_Re
```

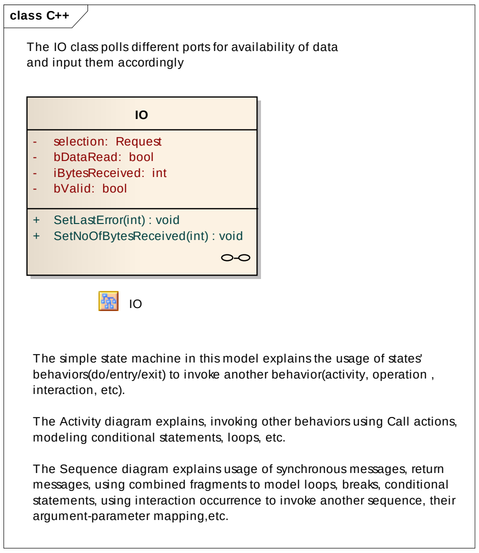
```

    }

    /* 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.


The IO class polls different ports for availability of data and input them accordingly

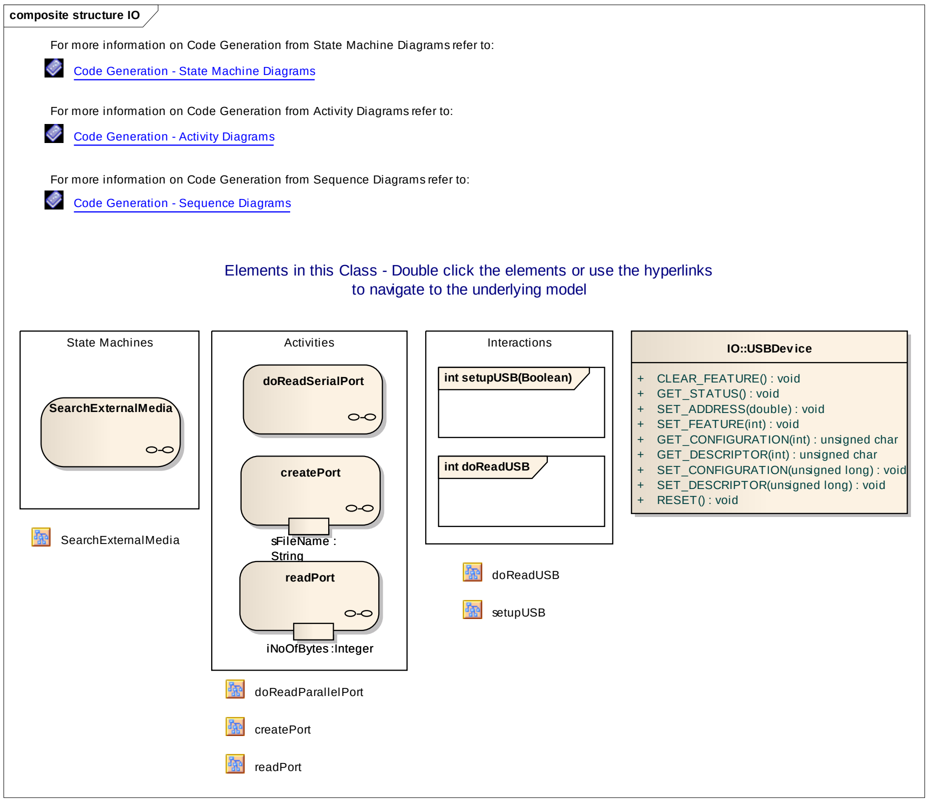
The simple state machine in this model explains the usage of states' behaviors(do/entry/exit) to invoke another behavior(activity, operation , interaction, etc).

The Activity diagram explains, invoking other behaviors using Call actions, modeling conditional statements, loops, etc.

The Sequence diagram explains usage of synchronous messages, return messages, using combined fragments to model loops, breaks, conditional statements, using interaction occurrence to invoke another sequence, their argument-parameter mapping,etc.

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.


For more information on Code Generation from State Machine Diagrams refer to:

Code Generation - State Machine Diagrams

For more information on Code Generation from Activity Diagrams refer to:

Code Generation - Activity Diagrams

For more information on Code Generation from Sequence Diagrams refer to:

Code Generation - Sequence Diagrams

Elements in this Class - Double click the elements or use the hyperlinks to navigate to the underlying model

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*.

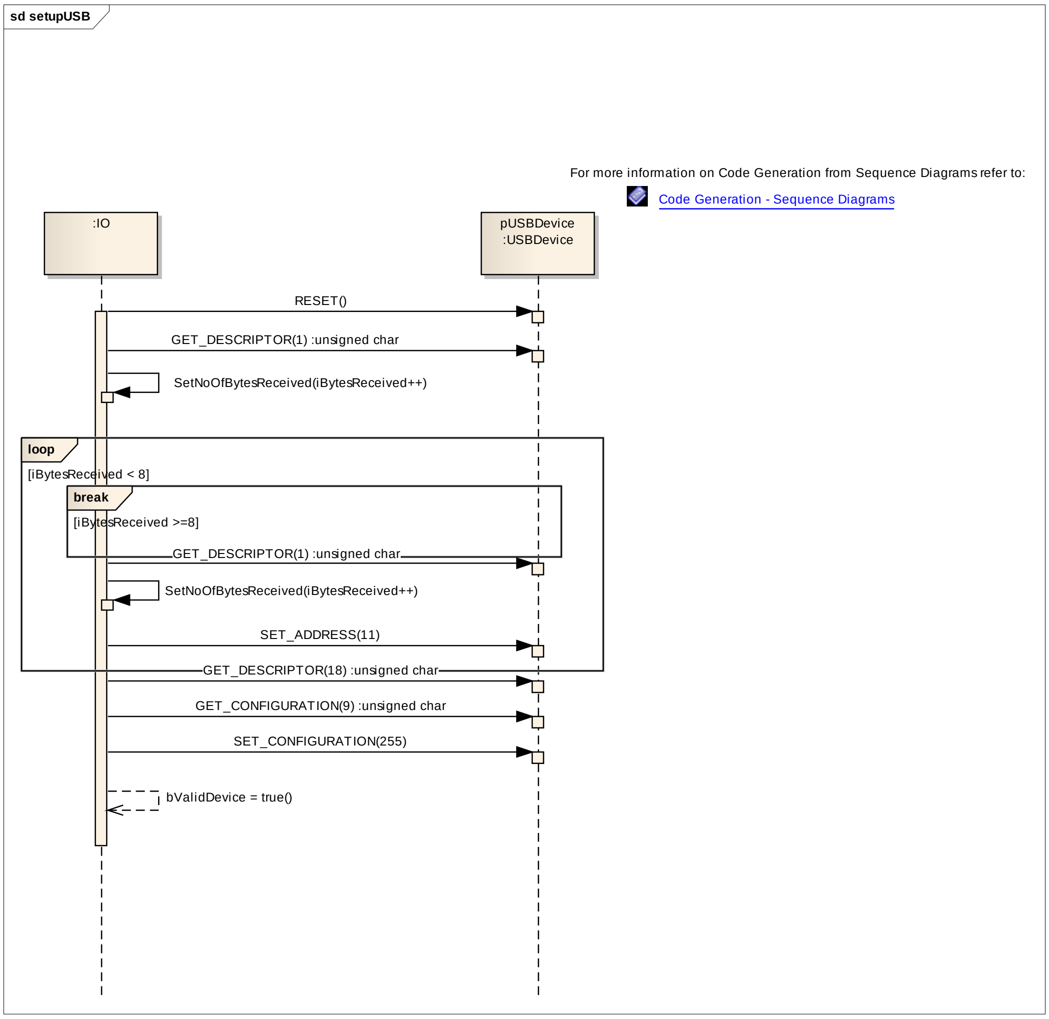

sd setupUSB
For more information on Code Generation from Sequence Diagrams refer to:
Code Generation - Sequence Diagrams
:IO
pUSBDevice :USBDevice
RESET()
GET_DESCRIPTOR(1) :unsigned char
SetNoOfBytesReceived(iBytesReceived++)
loop
[iBytesReceived < 8]
break
[iBytesReceived >=8]
GET_DESCRIPTOR(1) :unsigned char
SetNoOfBytesReceived(iBytesReceived++)
SET_ADDRESS(11)
GET_DESCRIPTOR(18) :unsigned char
GET_CONFIGURATION(9) :unsigned char
SET_CONFIGURATION(255)
bValidDevice = true()

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

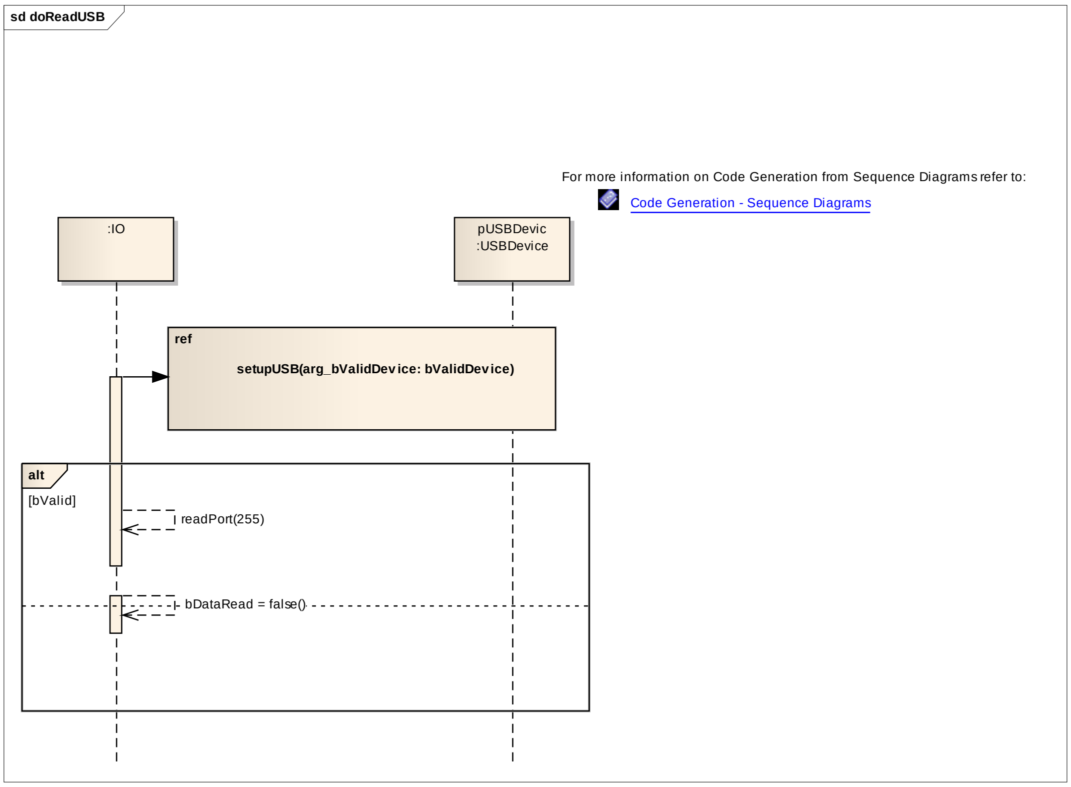
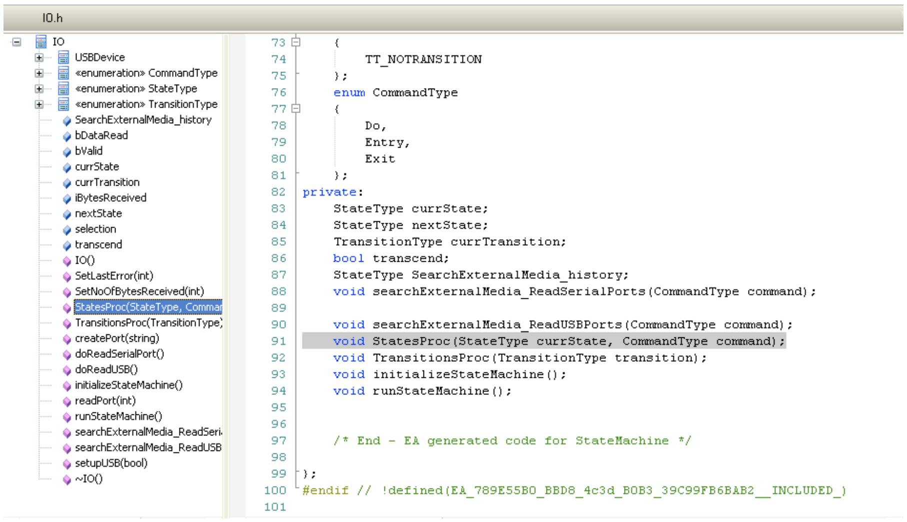

sd doReadUSB
For more information on Code Generation from Sequence Diagrams refer to:
Code Generation - Sequence Diagrams
:IO
pUSBDevic :USBDevice
ref
setupUSB(arg_bValidDevice: bValidDevice)
alt
[bValid]
readPort(255)
bDataRead = false()

Figure 13 – Reading the USB Port



```
    {
        TT_NOTRANSITION
    };
    enum CommandType
    {
        Do,
        Entry,
        Exit
    };
private:
    StateType currState;
    StateType nextState;
    TransitionType currTransition;
    bool transcend;
    StateType SearchExternalMedia_history;
    void searchExternalMedia_ReadSerialPorts(CommandType command);

    void searchExternalMedia_ReadUSBPorts(CommandType command);
    void StatesProc(StateType currState, CommandType command);
    void TransitionsProc(TransitionType transition);
    void initializeStateMachine();
    void runStateMachine();


    /* End - EA generated code for StateMachine */

};
#endif // !defined(EA_789E55B0_BBD8_4c3d_B0B3_39C99FB6BAB2__INCLUDED_)
```

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).

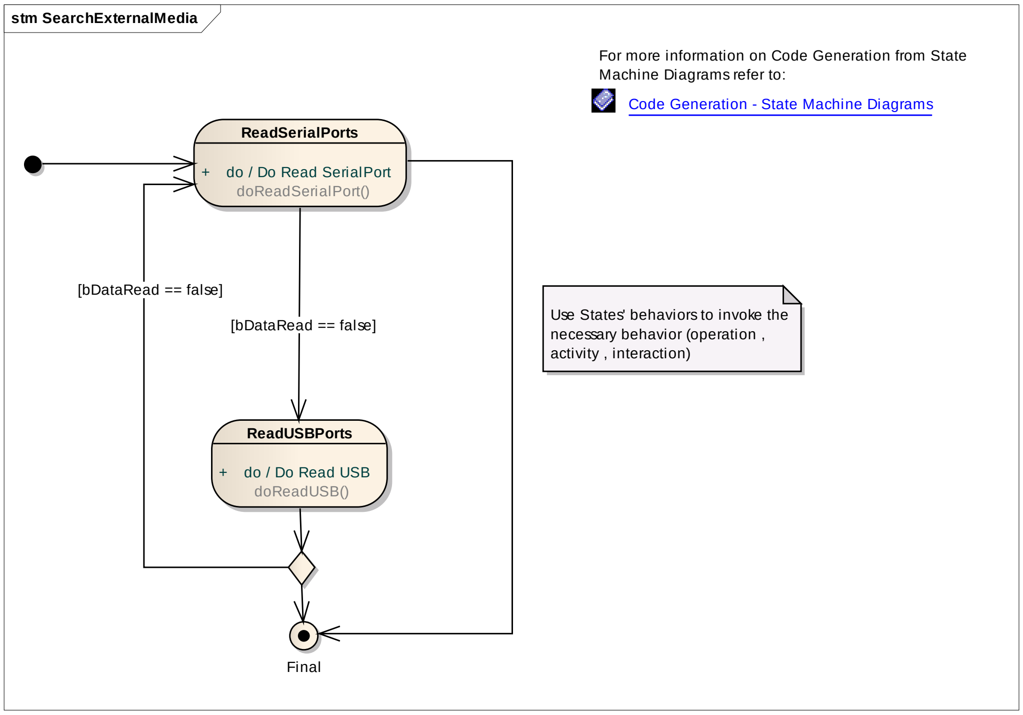
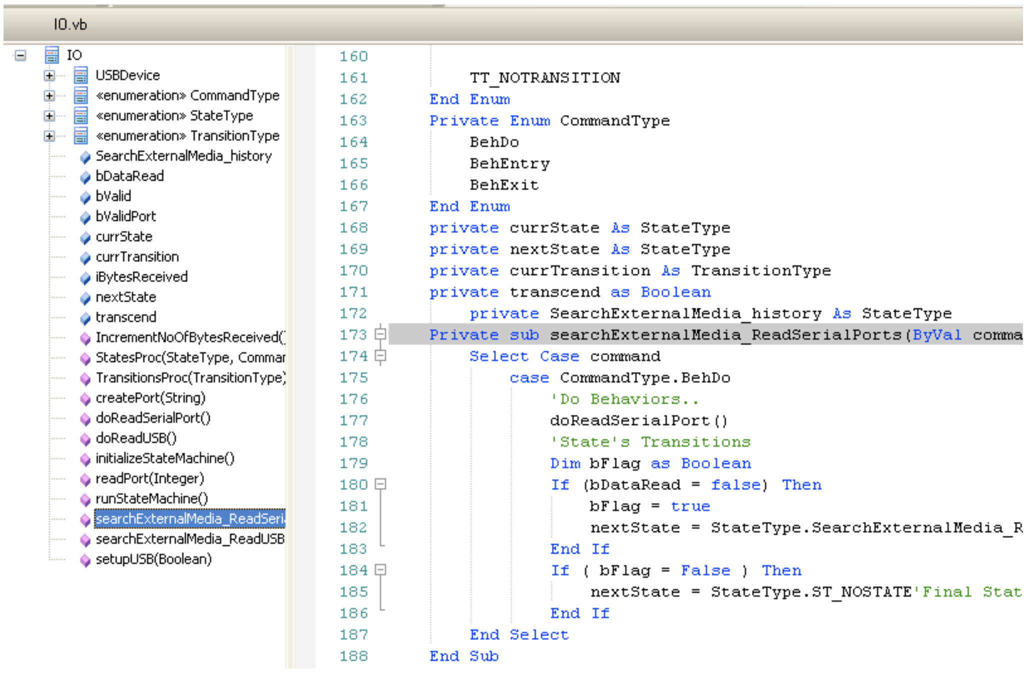

stm SearchExternalMedia
For more information on Code Generation from State Machine Diagrams refer to:
Code Generation - State Machine Diagrams
ReadSerialPorts
+ do / Do Read SerialPort
doReadSerialPort()
[bDataRead == false]
[bDataRead == false]
Use States' behaviors to invoke the necessary behavior (operation , activity , interaction)
ReadUSBPorts
+ do / Do Read USB
doReadUSB()
Final

Figure 15 – State Machine for Searching External Media



```
        TT_NOTRANSITION
    End Enum
    Private Enum CommandType
        BehDo
        BehEntry
        BehExit
    End Enum
    private currState As StateType
    private nextState As StateType
    private currTransition As TransitionType
    private transcend as Boolean
        private SearchExternalMedia_history As StateType
    Private sub searchExternalMedia_ReadSerialPorts(ByVal comma
        Select Case command
            case CommandType.BehDo
                'Do Behaviors..
                doReadSerialPort()
                'State's Transitions
                Dim bFlag as Boolean
                If (bDataRead = false) Then
                    bFlag = true
                    nextState = StateType.SearchExternalMedia_R
                End If
                If ( bFlag = False ) Then
                    nextState = StateType.ST_NOSTATE'Final Stat
                End If
        End Select
    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.

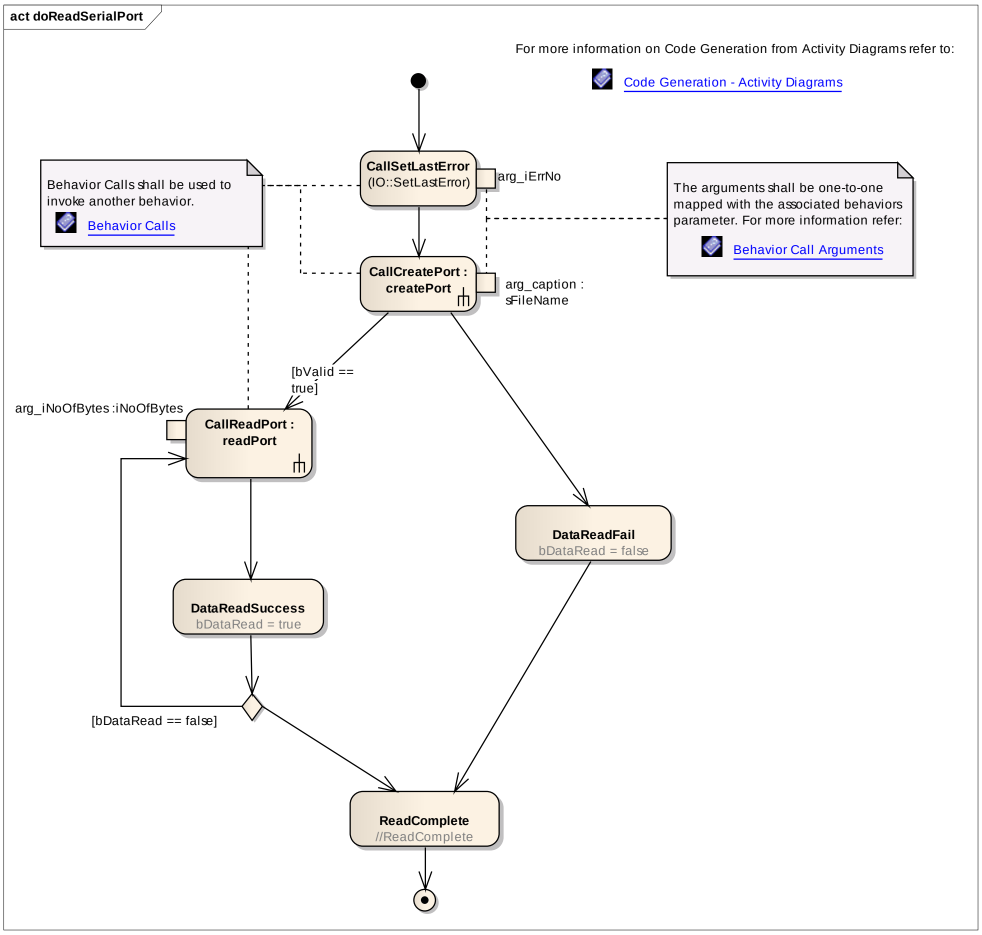
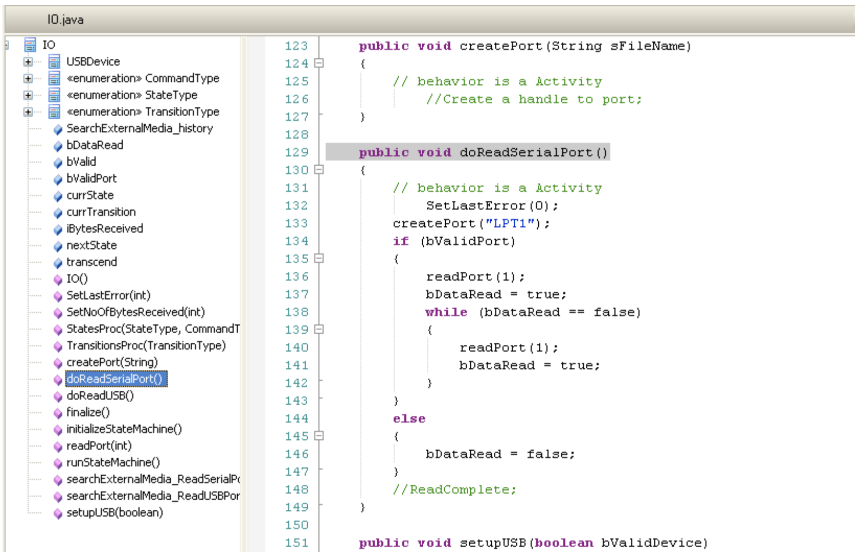

act doReadSerialPort
For more information on Code Generation from Activity Diagrams refer to:
Code Generation - Activity Diagrams
Behavior Calls shall be used to invoke another behavior.
Behavior Calls
CallSetLastError (IO::SetLastError)
arg_iErrNo
The arguments shall be one-to-one mapped with the associated behaviors parameter. For more information refer:
Behavior Call Arguments
CallCreatePort : createPort
arg_caption : sFileName
[bValid == true]
arg_iNoOfBytes :iNoOfBytes
CallReadPort : readPort
DataReadFail
bDataRead = false
DataReadSuccess
bDataRead = true
[bDataRead == false]
ReadComplete
//ReadComplete

Figure 17 – Activity Diagram for Reading the Serial Port



```
IO
  USBDevice
  «enumeration» CommandType
  «enumeration» StateType
  «enumeration» TransitionType
  SearchExternalMedia_history
  bDataRead
  bValid
  bValidPort
  currState
  currTransition
  iBytesReceived
  nextState
  transcend
  IO()
  SetLastError(int)
  SetNoOfBytesReceived(int)
  StatesProc(StateType, CommandT
  TransitionsProc(TransitionType)
  createPort(String)
  doReadSerialPort()
  doReadUSB()
  finalize()
  initializeStateMachine()
  readPort(int)
  runStateMachine()
  searchExternalMedia_ReadSerialP
  searchExternalMedia_ReadUSBPor
  setupUSB(boolean)
```

```java
public void createPort(String sFileName)
{
    // behavior is a Activity
        //Create a handle to port;
}

public void doReadSerialPort()
{
    // behavior is a Activity
        SetLastError(0);
    createPort("LPT1");
    if (bValidPort)
    {
        readPort(1);
        bDataRead = true;
        while (bDataRead == false)
        {
            readPort(1);
            bDataRead = true;
        }
    }
    else
    {
        bDataRead = false;
    }
    //ReadComplete;
}

public void setupUSB(boolean bValidDevice)
```

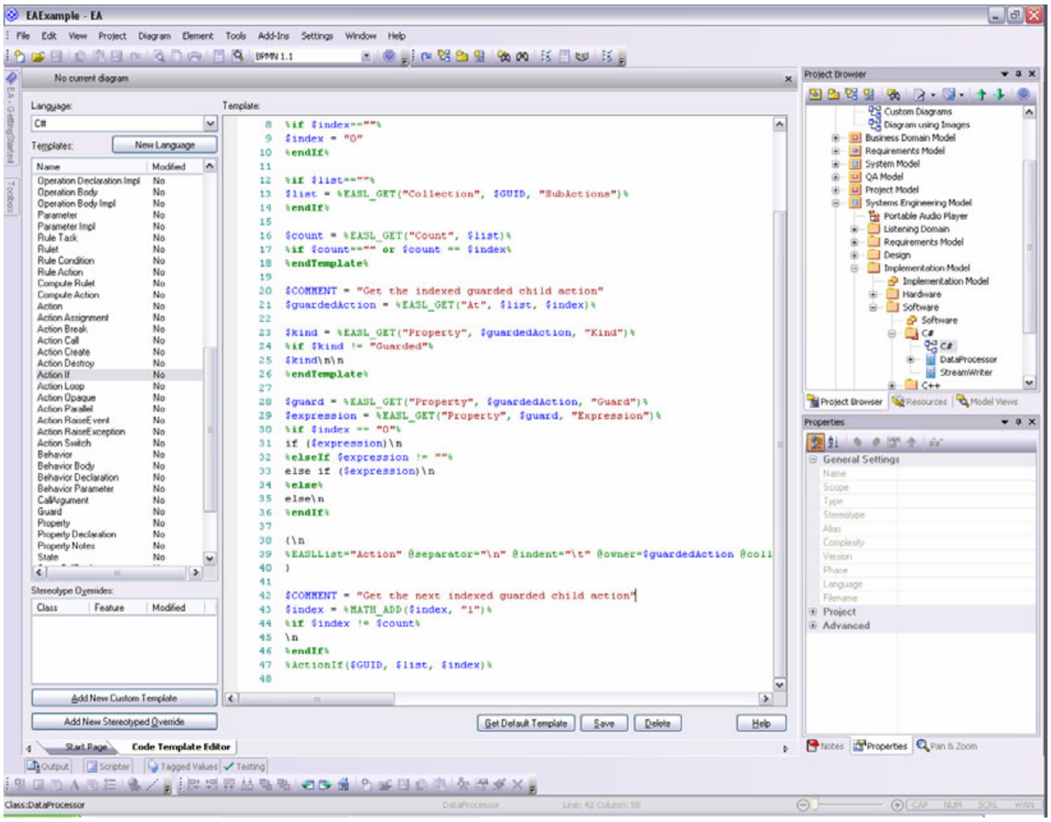
Figure 18 – Behaviorally generated Java code for Read Serial Port

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.



Language: C#

Name	Modified
Operation Declaration Impl	No
Operation Body	No
Operation Body Impl	No
Parameter	No
Parameter Impl	No
Rule Task	No
Rulet	No
Rule Condition	No
Rule Action	No
Compute Rulet	No
Compute Action	No
Action	No
Action Assignment	No
Action Break	No
Action Call	No
Action Create	No
Action Destroy	No
Action If	No
Action Loop	No
Action Opaque	No
Action Parallel	No
Action RaiseEvent	No
Action RaiseException	No
Action Switch	No
Behavior	No
Behavior Body	No
Behavior Declaration	No
Behavior Parameter	No
CallArgument	No
Guard	No
Property	No
Property Declaration	No
Property Notes	No
State	No

Template:

```
%if $index==""%
$index = "0"
%endIf%

%if $list==""%
$list = %EASL_GET("Collection", $GUID, "SubActions")%
%endIf%

$count = %EASL_GET("Count", $list)%
%if $count=="" or $count == $index%
%endTemplate%

$COMMENT = "Get the indexed guarded child action"
$guardedAction = %EASL_GET("At", $list, $index)%

$kind = %EASL_GET("Property", $guardedAction, "Kind")%
%if $kind != "Guarded"%
$kind\n\n
%endTemplate%

$guard = %EASL_GET("Property", $guardedAction, "Guard")%
$expression = %EASL_GET("Property", $guard, "Expression")%
%if $index == "0"%
if ($expression)\n
%elseIf $expression != ""%
else if ($expression)\n
%else%
else\n
%endIf%

{\n
%EASLList="Action" @separator="\n" @indent="\t" @owner=$guardedAction @coll
}

$COMMENT = "Get the next indexed guarded child action"
$index = %MATH_ADD($index, "1")%
%if $index != $count%
\n
%endIf%
%ActionIf($GUID, $list, $index)%
```

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

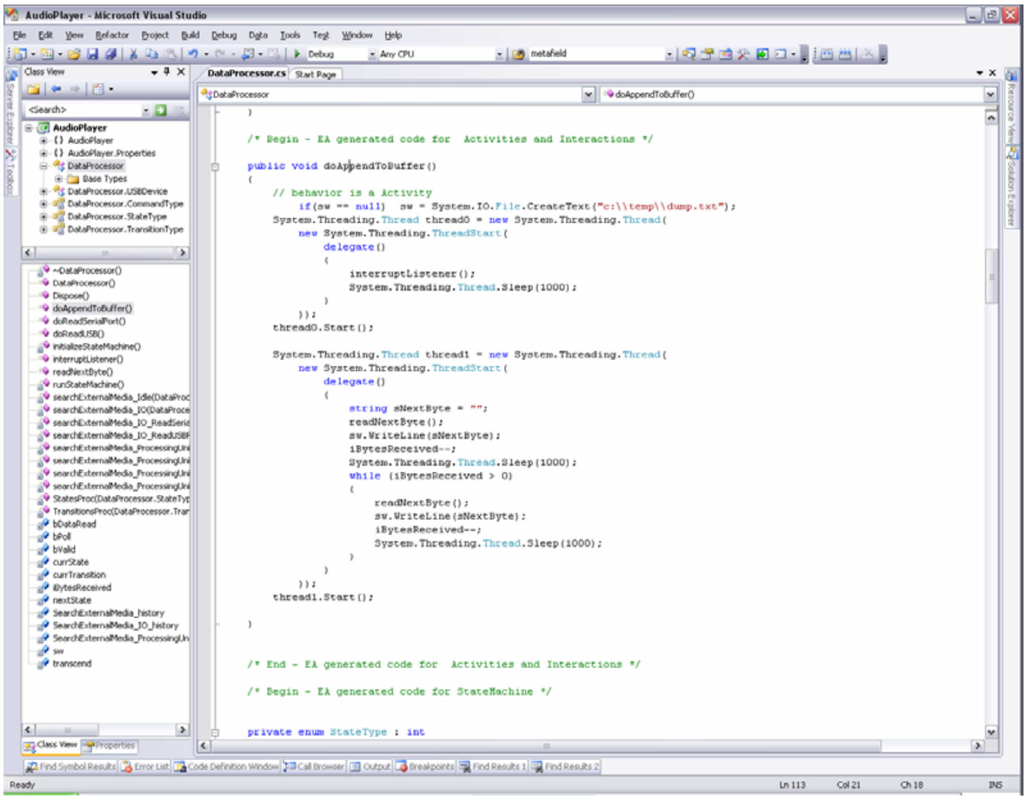
Templates can contain any or all of these constructs.

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.



```
    }

    /* 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.txt");
        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)
                    {
                        readNextByte();
                        sw.WriteLine(sNextByte);
                        iBytesReceived--;
                        System.Threading.Thread.Sleep(1000);
                    }
                }
            ));
        thread1.Start();

    }


    /* End - EA generated code for  Activities and Interactions */

    /* Begin - EA generated code for StateMachine */


    private enum StateType : int
```

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, or explore our "SysML JumpStart Training" on our website.

We wish you success in your development efforts!
