TEAMCENTER

Wiring Harness Design Tools Integration With Teamcenter

Teamcenter 2412

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

Support Center: **support.sw.siemens.com**

Send Feedback on Documentation: **support.sw.siemens.com/doc_feedback_form**

Contents

Getting started with Wiring Harness design tools integration

Wiring harness design tools integration overview 1-1
Wiring Harness Design Tools Integration prerequisites 1-1
Installing Wiring Harness Design Tools Integration using Deployment Center 1-2
Wiring Harness Design Tools Integration interface 1-3
Basic concepts 1-3
- Why use Wiring Harness Design Tools Integration 1-3
- Wire harness life cycle 1-4
- Wire harness integration architecture 1-5

Manage a wire harness in Teamcenter 1-7
- Basic tasks 1-7
- Creating a wire harness 1-7
- Revising a wire harness 1-10
- Exchanging data with ECAD and MCAD applications 1-11
- Use workflow to align status of a Capital tool electrical design with corresponding Teamcenter design revision 1-12
- Use a workflow to create a new Teamcenter design revision and its corresponding electrical design revision in the Capital tool 1-14

Wiring harness data model

About wiring harness data model 2-1
STEP AP212 and KBL models 2-3
Wire harness object model 2-5
- Electrical functional model 2-5
- Electrical logical model 2-6
- Electrical physical model 2-7

Wire harness objects 2-8
- Teamcenter wire harness objects 2-8
- Electrical components 2-8
- Signals and process variables 2-9
- Electrical interfaces 2-9
- Electrical connections 2-10
- Routes 2-11
- Allocations 2-12
- Relationships 2-13
- Reference designators 2-14

Allocations

About allocations 3-1
Allocation properties 3-2
Working with allocations 3-3

Creating allocation maps 3-3
Creating allocations 3-4

Configuring and administering Wiring Harness Design Tools Integration

Configuring and administering Wiring Harness Design Tools Integration 4-1
Administering PLM XML data transfers 4-1
Configuring for MCAD 4-17
Configure system for information exchange 4-17
Creating an application interface business object 4-17
Configure Application Ref for export 4-17
Working with sample SOA wire harness structure 4-18
About sample SOA wire harness structure 4-18
C++ sample client 4-18
Java sample SOA wire harness client 4-19
Administering electromechanical objects 4-20
Administering functionality 4-20
Administering connections 4-23
Administering signals 4-25
Administering item elements 4-26
Setting user exits 4-29
Using a Multi-Site environment 4-30
Identifying user name and password for Capital tool using custom exit 4-31

Wiring Harness APIs

Wire Harness APIs 5-1
PSCONN module 5-1
GDE module 5-2
SIGNAL module 5-3
ROUTE module 5-4
ALLOC module 5-5

Integrating with electromechanical applications

Integrating with electromechanical applications 6-1
Harness design process 6-1
Harness design process flow 6-1
Authoring electrical connectivity information in ECAD 6-2
Publishing electrical design data to Teamcenter 6-4
Working with the data in Teamcenter 6-5
Sharing ECAD data with other applications 6-6
Reconciling the Teamcenter data with additional electrical information 6-8

1. Getting started with Wiring Harness design tools integration

Wiring harness design tools integration overview

Teamcenter Wiring Harness Design Tools Integration is a seamless integration of logical and physical design processes, route generation, wire length determination, and design validation. It manages the design data, the electrical connectivity data, and component information using intelligent business objects. It is a secure, single source of product and process as well as systems engineering information; requirements, workflow, change, and BOM management; component reuse; and multiple variant configurations.

Wiring Harness Design Tools Integration provides extensions to the standard Teamcenter data model to support external design tools. The integration supports functional and physical models, connections, signals, and their associated relationships. It allows you to integrate your existing ECAD or MCAD tools with Teamcenter to exchange data.

Teamcenter provides a comprehensive wire harness life cycle management solution that extends from initial inception through creation, analysis, manufacturing, service, and end-of-life disposition. The solution enables electrical, mechanical, and manufacturing design teams to collaborate on wire harness designs.

Wiring Harness Design Tools Integration prerequisites

Prerequisites	• You must have a Teamcenter license that enables authoring to use Wiring Harness Design Tools Integration. • Make sure you have installed the following: • An ECAD and/or an MCAD application that supports harness design. • Teamcenter four-tier setup (for integration using SOA). • Business Modeler IDE for configuring the integration.
Enable Wiring Harness Design Tools Integration	**Install Wiring Harness Design Tools Integration**.

Configure Wiring Harness Design Tools Integration	When you install Wiring Harness Design Tools Integration, all necessary harness objects, datasets, electrical design, and electrical harness BOM views are installed. Also, the preferences are set and closure rules installed.
Start Wiring Harness Design Tools Integration	You can start Wiring Harness Design Tools Integration either manually (by importing the PLM XML files) or automatically (using SOA).

Installing Wiring Harness Design Tools Integration using Deployment Center

Add the Wiring Harness Design Tools Integration application to your existing Teamcenter environment.

Procedure

1. Log on to Deployment Center and select the environment to which you want to add Wiring Harness Design Tools Integration.

2. Go to the **Applications** task. Click **Add or Remove Selected Applications**.

3. In the **Available Applications** panel, use the web browser search to find the following applications:

   - **Electrical and Wire Harness Configuration**

     Installs the wiring harness functionality.

   - **Multi-Disciplinary Associations**

     Installs the Multidisciplinary Associations feature for collaboration between various disciplines during the product design phase.

   Select the applications, and then click **Update Selected Applications**.

   Deployment Center automatically selects any additional dependent applications.

4. Go to the **Components** task.

5. In the **Selected Components** list, note any remaining components whose configuration status is not **100%**. Select each incomplete component, enter required parameters, and save component settings until all components in the environment show a configuration status of **100%**.

   When all components are fully configured, the **Deploy** task is enabled.

6. Go to the **Deploy** task. Click **Generate Install Scripts** to generate deployment scripts you will use to update affected machines.

When script generation is complete, note any special instructions in the **Deploy Instructions** panel.

7. Locate deployment scripts, copy each script to its target machine, and then run each script on its target machine.

   For more information about running deployment scripts, see *Deployment Center — Usage*.

Wiring Harness Design Tools Integration interface

Teamcenter does not provide dedicated applications or user interfaces to manage electrotechnical information. You can view and manage your data in the following Teamcenter applications:

- My Teamcenter

  Teamcenter Basics allows you to create, revise, and delete item elements, revisable and nonrevisable connections, and signal objects.

- Structure Manager

  Structure Management on Rich Client — Usage allows you to:

  - Compare product structures (BOMs) that include design objects to identify differences.
  - Connect or disconnect a connection.
  - Create, paste, remove, and view notes for an item element.

- Multi-Structure Manager

  Multi-Structure Management allows you to create allocations between different views.

Basic concepts

Why use Wiring Harness Design Tools Integration

Electromechanical products have complex electrical and mechanical systems. Each system has a number of variants, each requiring a different configuration of the wire harness. It is a challenge to design wire harnesses as each configuration must fit correctly within the mechanical framework of the product. Many products require some type of physical connection between their internal components or connection between one piece of electronic hardware and another device. Maintaining development schedules, costs, quality, and product reliability targets while developing this interconnect requires the wire harness life cycle to be integrated with the electronic and mechanical stages of the design process.

The electrical system in an electromechanical product evolves as modifications to the original harness design are required. To effectively manage the design changes throughout its life cycle and to have

enterprise-wide visibility and sharing of data, you need to integrate the design process in the product life cycle management system. Teamcenter provides the Wiring Harness Design Tools Integration framework, which streamlines the design process by managing the life cycle of various electrical components, connectors, harnesses, clips, fixtures, and signals along with design data.

Wiring Harness Design Tools Integration accelerates the development schedules through an integrated logical/physical design flow and improves quality through systems engineering and requirements management. This helps to decrease design cost through wire harness design reuse and improve reliability by using an integrated design and verification flow.

To understand how to make the best use of Wiring Harness Design Tools Integration, you should be familiar with the wire harness data model, features, life cycle, and architecture. In addition, an understanding of the integration's data model and SOA integration is helpful.

Wire harness life cycle

The wire harness life cycle:

- Leverages Teamcenter requirements management, workflow management, configuration management, change control, and manufacturing management functionalities.
- Enables design teams to accelerate time-to-market and reduce development cost.
- Manages the structure and dependencies.
- Tracks electrical/mechanical data: connections, signals, ports, routing, topology, wire type, length, connectors, wrap, and plugs, and so on.
- Allocates and associates components across all phases of development.

The entire design process for large electromechanical products typically follows systems engineering methodology, where the product is decomposed into functionally partitioned systems. This process can be easily managed and designed as explained here.

1. Input and specifications: Covers the general requirements applicable for electrical wire harnesses such as design, quality, manufacturing standards, product features, performance, and cost considerations.
2. Requirements: Consists of identifying target requirements for the product to be designed. These include design requirements from several domains, including electrical domain, manufacturing requirements, and other requirements such as safety and environmental regulations.
3. System design: Covers the requirements along with options and variants to be offered for the product. Either existing systems, which meet the target requirements are identified, or new systems are designed to meet any requirements not addressed by existing systems. Electrical system design also includes defining the electrical connectors and the electrical interfaces between the systems.

4. Functional/Behavioral model: Comprises product requirements, use cases, and logical equations. A functional model provides a breakdown of various functional units of the product and signal interactions among them in the functional network. A behavioral model consists of analytical representation of various components, such as interfaces and devices, along with state diagrams.

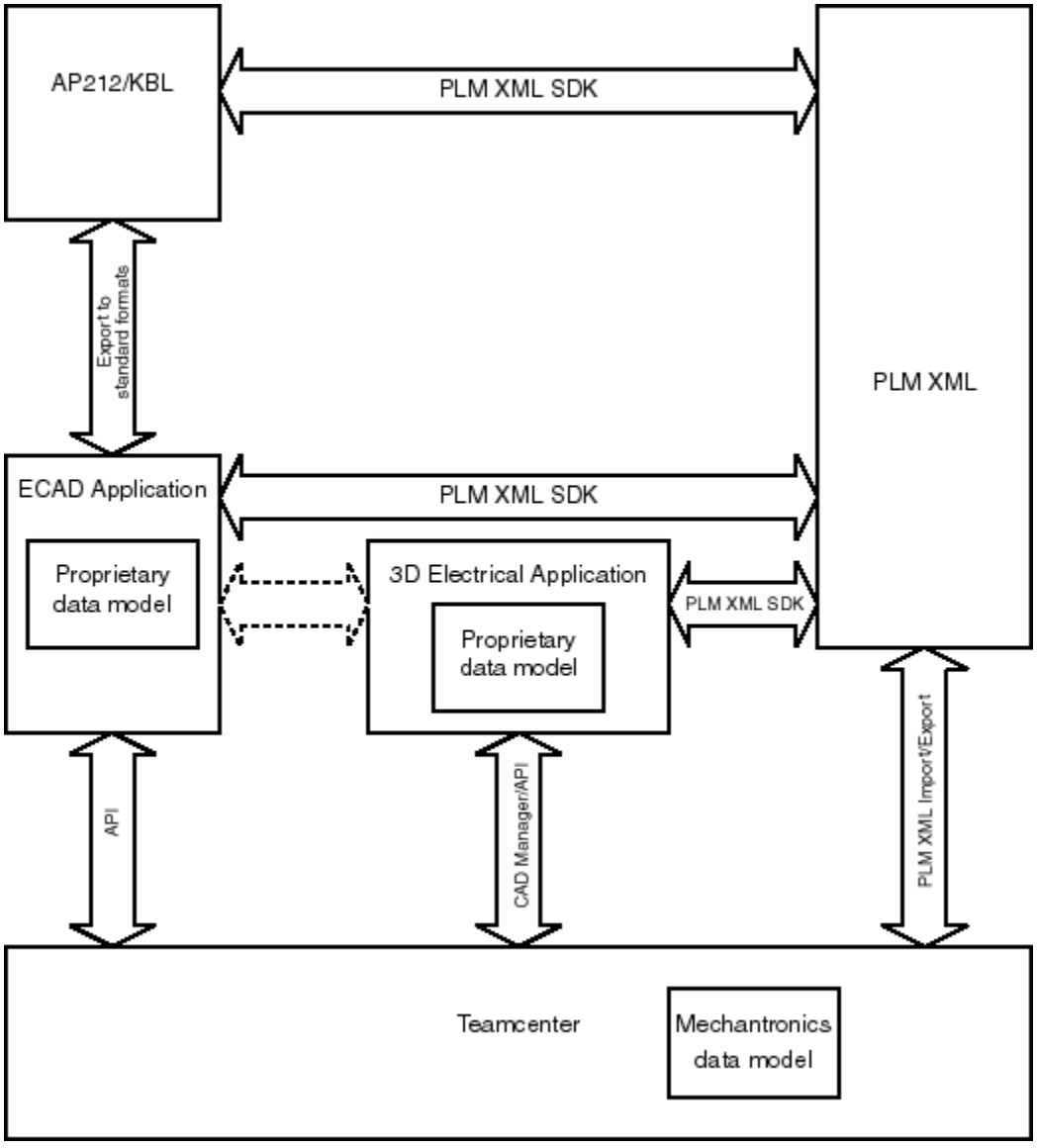
5. Simulation and analysis: Consists of the functional or behavioral model. The primary purpose of building a functional or behavioral model for an electrical system is to further perform various simulations and studies on this model early in the design cycle. Simulation and various types of analysis help check the validity of a wire harness design at any stage in the process.

Wire harness integration architecture

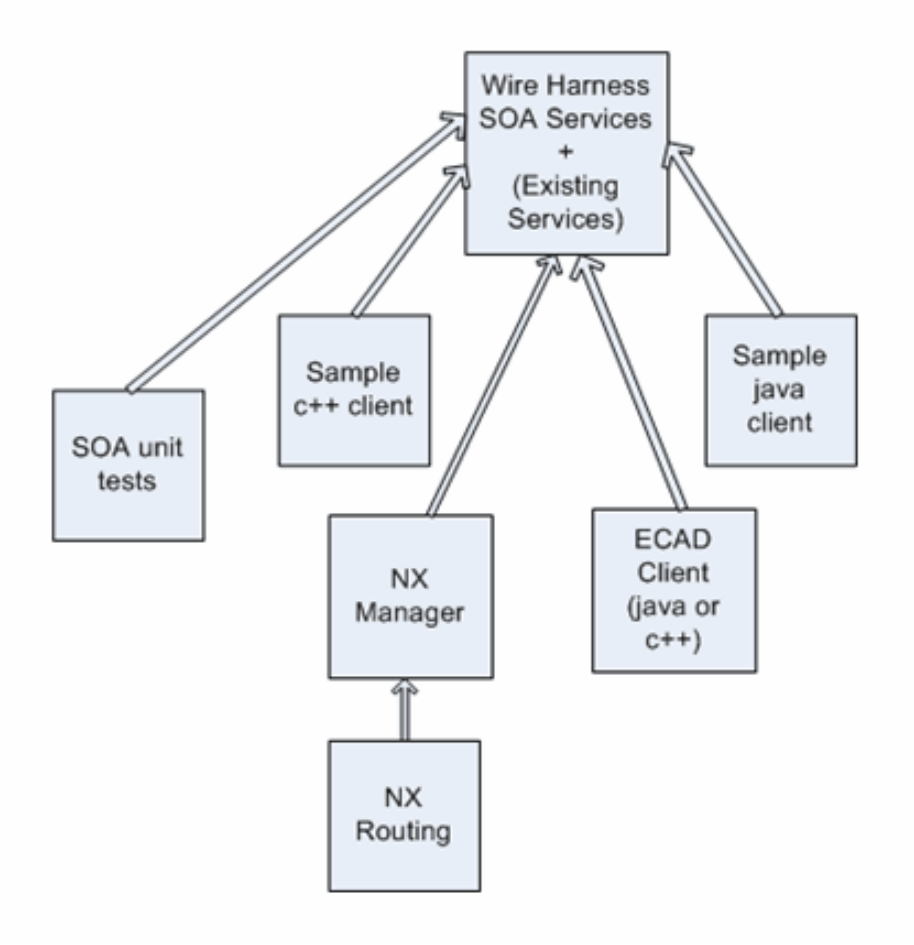
Wiring Harness Design Tools Integration provides an overlay to the standard Teamcenter data model. This overlay expands the Teamcenter schema, providing additional data objects and API functions that allow you to define and manage electrotechnical information. The Teamcenter model supports AP212 and KBL representations of electromechanical data.

You can exchange design data between Teamcenter and the external design tools in the industry-standard PLM XML format. The PLM XML schema supports representations of electromechanical data by KBL elements, AP212 elements, or a combination of both standards. In general, if the electromechanical data modeled by an ECAD application can be exported in the AP212 model, it can also be unambiguously exported as PLM XML or in the wire harness design model (both the model and the PLM XML schema are AP212 compliant).

The following figure shows the general arrangement of the Wiring Harness Design Tools Integration.



You can also exchange design data using SOA. The following figure shows the Wiring Harness Design Tools Integration using SOA.



Manage a wire harness in Teamcenter

Basic tasks

To manage a typical wire harness in the Teamcenter environment, you can:

- Create a wire harness.
- Revise a wire harness.
- Exchange data with ECAD and NX applications.

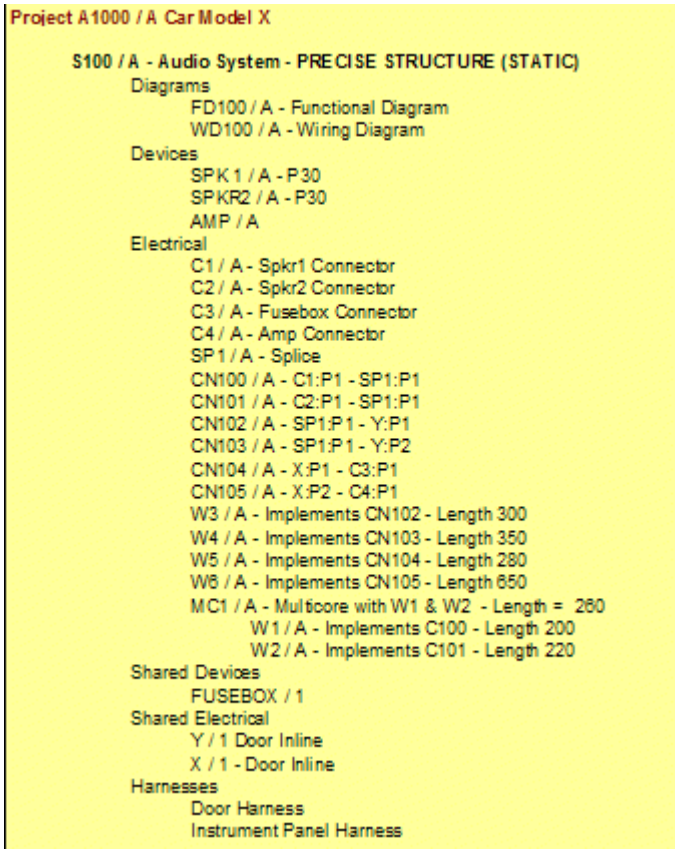
Creating a wire harness

You can use the Teamcenter functionality to create and manage wire harnesses in several ways, depending on your business practices and environment. A typical scenario follows:

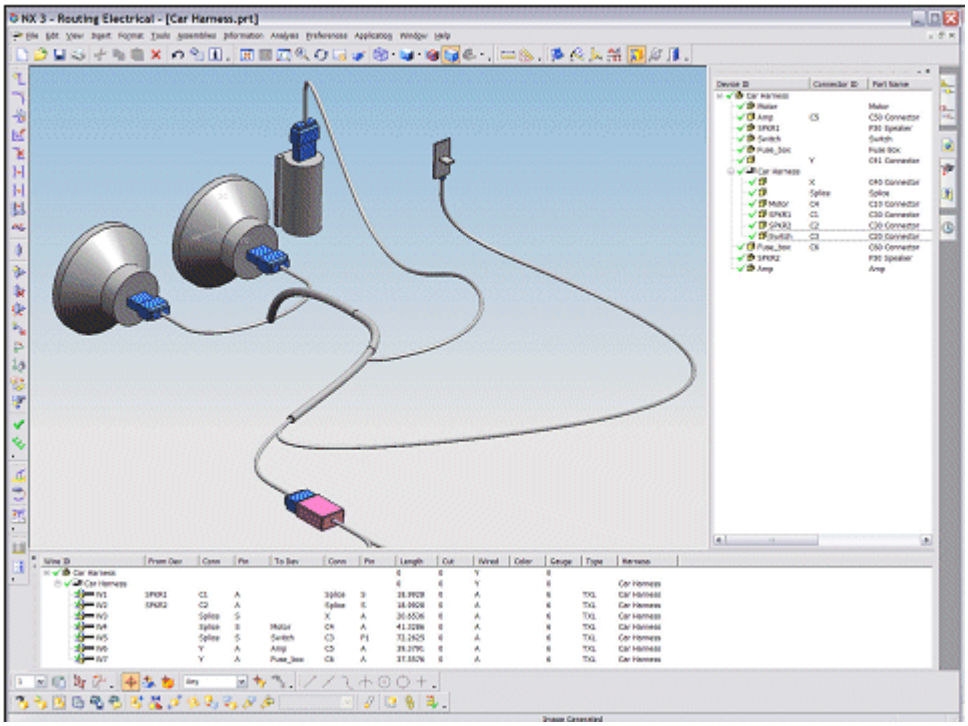
1. The electrical designer creates the wiring designs in the wiring design application. Each design typically defines a system.

2. The electrical designer publishes the designs for review, or triggers a data exchange with the 3D MCAD system (for example, NX). The electrical designer changes the release status of the designs in the wiring design application, publishing the data into Teamcenter. The electrical designer may publish a single wiring diagram, a harness, or the contents of a build list.

   For each published design, Teamcenter publishes the structure into an electrical view within the target Teamcenter product structure. This structure is typically in a BOM-like format, as shown in the following figure.



3. The mechanical designer instantiates devices (physical devices) and connectors in a 3D space, possibly in the context of a 3D shape such as a vehicle. The mechanical designer also creates path segments to develop a harness structure or to reserve space for a harness. The following figure shows a typical physical harness design developed in NX.



4. The mechanical designer is notified that the electrical view of his product structure has been updated in Teamcenter. The mechanical designer accesses splices and devices (defined in the ECAD system) and connections (defined in the electrical system) from the electrical view.

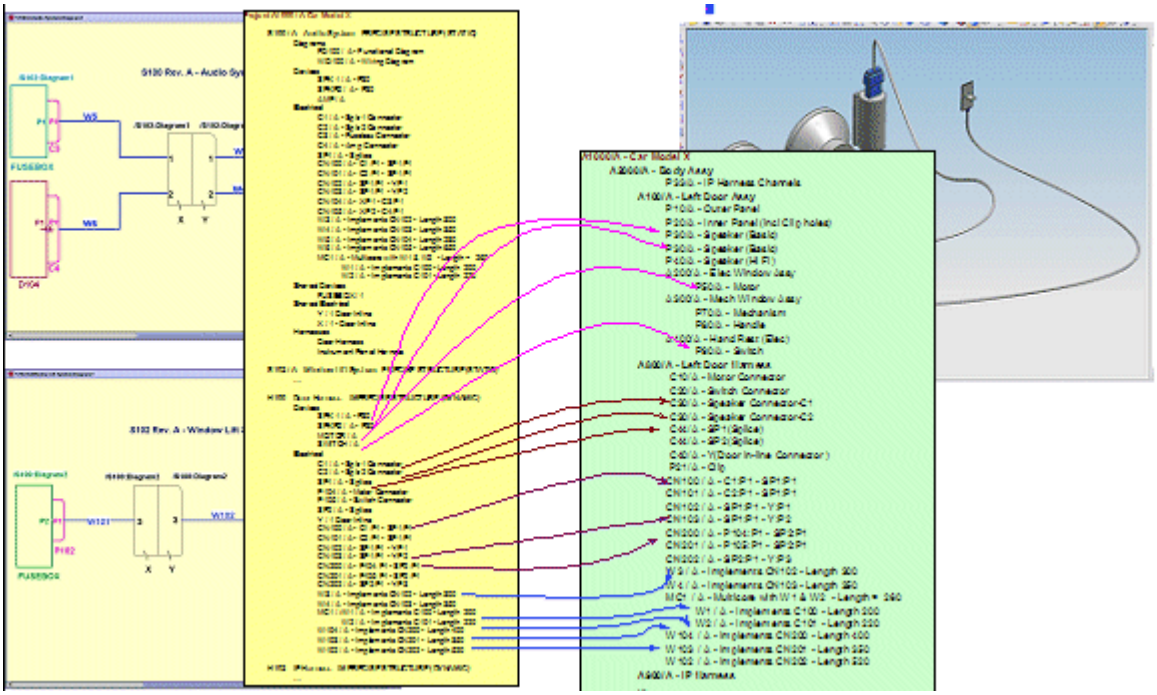
   The mechanical designer may also optionally access electrical connectors from the electrical view. This may be unnecessary for some scenarios.

5. In the 3D MCAD application, the mechanical designer allocates electrical devices to physical connectors. Allocations may be necessary for some or all connectors, depending on the scenario.

   At this point, the mechanical designer may optionally publish the physical view of the harness to Teamcenter, with the allocations to the electrical view. This action unifies the electrical and physical product structures, making them available in Teamcenter. Routing is not required at this stage.

6. In the 3D MCAD application, the mechanical designer routes connections and, in doing so, creates routes for electrical connections.

7. When ready, the mechanical designer publishes the physical view of the harness to Teamcenter. The following figure shows the allocations for the example design. At this point, electrical wires and multicores are updated with routes and lengths.



8. The electrical designer is notified that the electrical view is updated in Teamcenter.

9. The electrical designer retrieves routes for connections from Teamcenter and updates the wire and multicore lengths in the wire harness application.

Revising a wire harness

You can use the Teamcenter functionality to revise existing wire harnesses in several ways, depending on your business practices and environment. A typical scenario follows:

1. The electrical designer makes some changes to an electrical design and creates a new revision.

2. When the electrical designer publishes the revised design, Teamcenter:

   - Publishes the design into a revised structure in the electrical view in the target Teamcenter product structure. A new structure is added for the revised system and changes are made to the existing harness structure.

   - Publishes diagrams in SVG format into the same electrical view.

3. The mechanical designer is notified that the electrical view of the product structure is updated in Teamcenter.

   The mechanical designer therefore accesses splices and devices (defined in the ECAD system) and connections (defined in the electrical system) from the electrical view.

4. In the 3D MCAD application, the mechanical designer allocates or reallocates electrical devices to physical devices, according to the changes made to the harness. It is not necessary to reallocate unchanged objects.

   At this point, the mechanical designer may publish the revised physical view of the harness to Teamcenter, with the allocations to the electrical view. This action unifies the electrical and physical product structures, making them available in Teamcenter. Routing is not required at this stage.

5. In the 3D MCAD application, the mechanical designer reroutes connections.

6. When ready, the mechanical designer publishes the revised physical view of the harness to Teamcenter. At this point, electrical wires and multicores are updated with routes and lengths.

7. The electrical designer is notified that the electrical view is updated in Teamcenter.

8. The electrical designer retrieves routes for connections from Teamcenter, and updates wire and multicore lengths in the wire harness application.

Making electrical changes

The electrical designer may make changes to an electrical design but may not create a new revision (These are work-in-progress changes to a design that is already published in Teamcenter). In this case, the electrical designer publishes the changes to the *same* revision of the objects in the electrical structure.

Making mechanical changes

Electrical objects are authored or revised only in the ECAD system. The mechanical designer can move devices and place wire protection on the routed wires, additional splices, and supporting accessories. If such changes occur, the electrical designer is notified that the electrical view has been updated in Teamcenter.

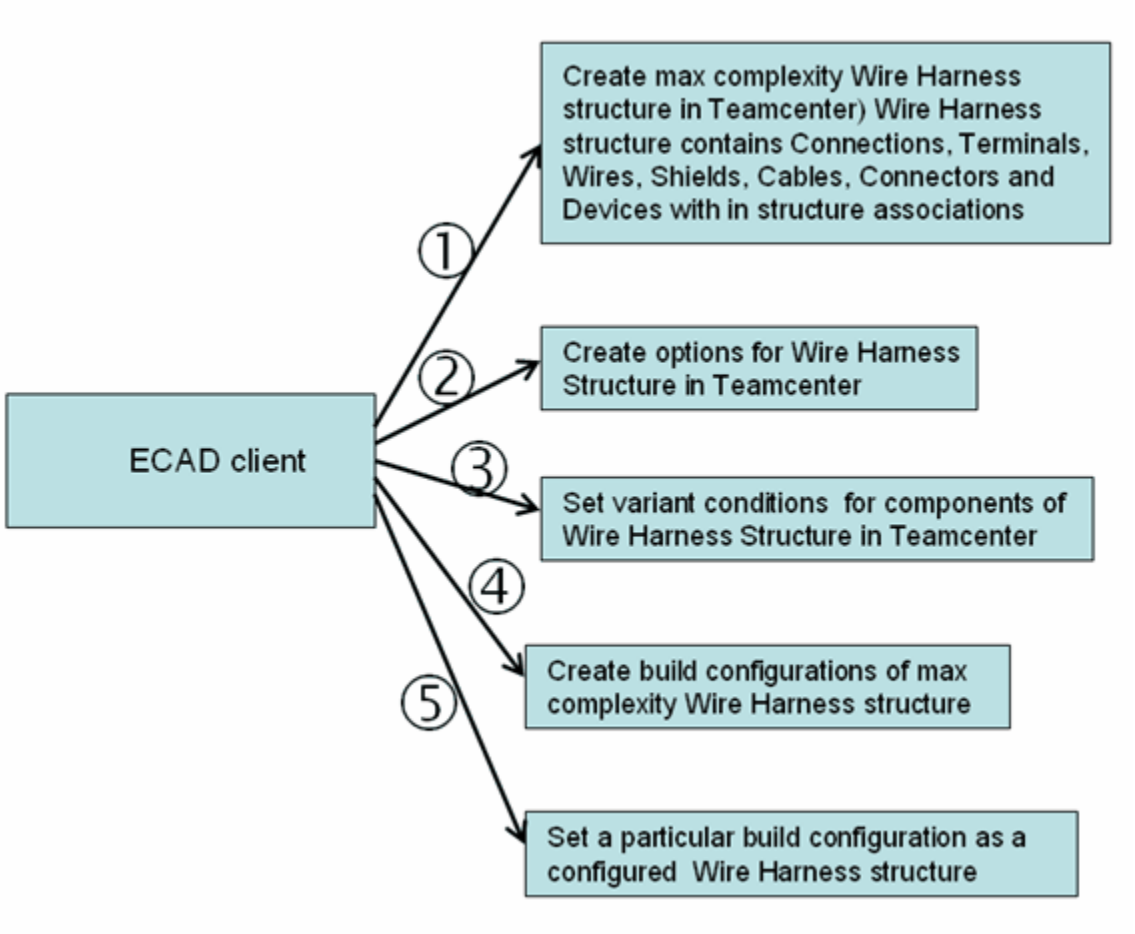
Exchanging data with ECAD and MCAD applications

Wire harness data exchange can be done using either PLM XML or SOA:

Teamcenter acts a central repository of master wire harness data and external ECAD and MCAD applications can create and modify the harness design data in Teamcenter. This enables seamless transfer of harness design between ECAD-Teamcenter-MCAD without data loss and redundancy.

You can exchange wire harness data, including options and variants associated with NX and ECAD with Teamcenter. ECAD and MCAD applications can exchange electrical information, and Teamcenter can capture design options created in electrical authoring applications. The SOA services facilitate data exchange with NX and ECAD applications.

The following figure shows the data exchange between an ECAD client and Teamcenter.



Use workflow to align status of a Capital tool electrical design with corresponding Teamcenter design revision

An electrical design published using the Capital tool and its corresponding design revision in Teamcenter can be reviewed using the **Electrical Design Review** workflow. Based on the outcome of the review process, the designs are either released or rejected.

This workflow process is typically initiated by an engineering user or a designer on an electrical design revision in Teamcenter, which is originally created or published using Capital.

The **Electrical Design Review** workflow template is provided out of the box. It is used to align the state between the Teamcenter designs and the corresponding Capital tool electrical designs.

For successful execution of **Electrical Design Review** workflow process, your Teamcenter administrator must set the following Teamcenter preferences:

- Set the value of the **Webservice_url_mentorCapital** preference to *http://host_ip:49901/chs/cis*. The IP address in the URL is the address of the host of Capital integration server.

- Set the value of **Webservice_verify_host_mentorCapital** preference to *false*. Set it to *true* if you want to use an SSL connection.
- Set the value of **Webservice_verify_peer_mentorCapital** preference to *false*. Set it to *true* if you want to use an SSL connection.
- Set the value of **Webservice_ssl_cert_file_mentorCapital** preference to the *SSL certificate(.pem)* file path if you want to use an SSL connection.

The workflow applies a Released or Rejected status on the Electrical designs and the Teamcenter designs. The following tasks are performed in the Electrical Design Review workflow.

1. **CapitalDesignReview**

   This task includes a design review process on the Teamcenter design revision. It consists of:

   a. The **Select-sign off** task to add the participants as the list of reviewers.

   b. The **Perform-sign off** task to review the design and either approve or reject the design.

2. **ApplyReleaseOnCapitalDesign/ApplyRejectOnCapitalDesign**

This task includes fetching the electrical designs from the Capital tool and applying a status of **Release** or **Reject** on the designs based on the outcome of the **CapitalDesignReview** task. It uses two action handlers:

- HRN-set-release-state action handlers to fetch the Capital designs and apply the **Release** status.
- HRN-set-reject-state action handlers to fetch the Capital designs and apply the **Reject** status.

3. **Released status/Rejected status**

   This task includes applying the **Released** status or the **Rejected** status on the Teamcenter design revision based on the outcome of the **ApplyReleaseOnCapitalDesign** task or the **ApplyRejectOnCapitalDesign** task.

4. **Notify/NotifyApproval/NotifyReleaseUpdateFailed/NotifyRejectUpdateFailed**

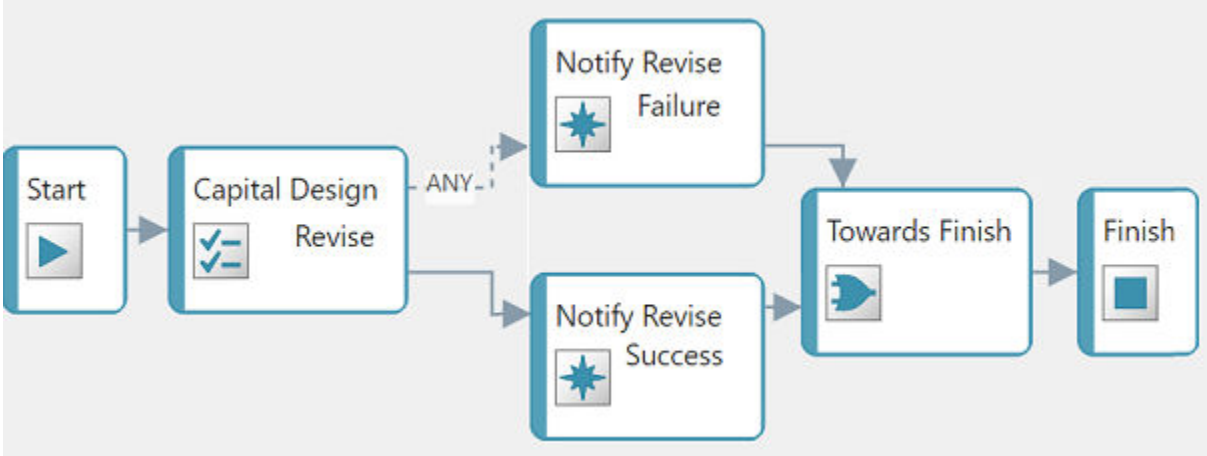
   These tasks provide Active Workspace notifications and SMTP notifications (for example, Outlook-based email notifications) for the different outcomes of the **Release status/Rejected status** task or the **ApplyReleaseonCapitalDesign/ApplyRejectOnCapitalDesign** task.

Use a workflow to create a new Teamcenter design revision and its corresponding electrical design revision in the Capital tool

An engineering user or designer can use the **Electrical Design Revise** workflow to create a new Teamcenter design revision of an electrical design published from the Capital tool and the corresponding new electrical design revision in the Capital tool. This workflow template is available out of the box.

When the Teamcenter design revision is submitted to the **Electrical Design Revise** workflow:

- A new revision of the design is created in Teamcenter.
- A new revision of the corresponding electrical design is created in the Capital tool.
- The new revision of the electrical design in the Capital tool gets associated to the new revision of the design in Teamcenter.



The **Electrical Design Revise** workflow consists of the following tasks:

- **CapitalDesignRevise**

  This task creates new revisions of the design in Teamcenter and the corresponding design in the *Capital* tool.

- **NotifyReviseFailure/NotifyReviseSuccess**

  These tasks provide Active workspace notifications as well as SMTP (for example, Outlook-based email) notifications for the different outcomes of the *Capital* design revision tasks.

2. Wiring harness data model

About wiring harness data model

The data model in Teamcenter allows support for many types of models and domain-specific features. It allows you to create different types of models such as system models, functional models, logical models, physical models, and manufacturing models. The use of BOM views in Teamcenter allows you to create multiple view types for each type of model. For example, a view type can be defined to represent the functional model and another view type can be defined to represent the physical model. Allocations can then be built to link various view types of a product to capture the associations between the components from one view to another. It also enables you to define and manage wire harnesses for products consisting of multiple options and variants.

By leveraging Wiring Harness Design Tools Integration as a single source of product and process knowledge and integrating both internally developed and third-party tools, you can reduce development costs and the time-to-market.

To effectively use this solution, you should start by familiarizing yourself with the following abstract classes. You can use these abstract classes to model Teamcenter Mechatronics Process Management objects:

- Item

  Items are generic, manageable, revisable, and releasable objects that can be used in a product structure. In a physical model, items can be used to represent parts. When integrated with a CAD system, items typically represent physical components instantiated in an assembly structure. When integrated with an ECAD system, items can be used to represent electrical components such as electrical devices, connectors, and wires. Items can also be used to represent system modules and electrical assemblies such as electrical harnesses consisting of connectors and wires and allow the users to model the system breakdown of a product.

  In functional models, items can be used to represent the functional breakdown of the product. In this case, an item can be used to represent a function. Teamcenter provides a predefined item type called **Functionality** for this purpose. You can also define custom types of an item to meet your specific needs.

  You can substitute mechatronics items by using the same item type or its subtypes for the **Substitute** operation. The following table lists the possible mechatronics items for the **Substitute** operation.

Object	Recommended types for Substitute
Functionality	**Functionality** or its subtype
Connection	**Connection** or its subtype
Network	**Network** or its subtype

Object	Recommended types for Substitute
PSConnection	**PSConnection** or its subtype
Signal	**Signal** or its subtype
PSSignal	**PSSignal** or its subtype
Message	**Message** or its subtype
HRN_Accessory	**HRN_Accessory** or its subtype
HRN_AssemblyPart	**HRN_AssemblyPart** or its subtype
HRN_CavityPlug	**HRN_CavityPlug** or its subtype
HRN_CavitySeal	**HRN_CavitySeal** or its subtype
HRN_CoPackPart	**HRN_CoPackPart** or its subtype
HRN_ConHousing	**HRN_ConHousing** or its subtype
HRN_Fixing	**HRN_Fixing** or its subtype
HRN_GenTerminal	**HRN_GenTerminal** or its subtype
HRN_GeneralWire	**HRN_GeneralWire** or its subtype
HRN_Harness	**HRN_Harness** or its subtype
HRN_Module	**HRN_Module** or its subtype
HRN_WireProtect	**HRN_WireProtect** or its subtype
HRN_Cable	**HRN_Cable** or its subtype
HRN_Shield	**HRN_Shield** or its subtype

- Item elements

  Item elements or generic design elements (GDE) are manageable, nonrevisable, and releasable objects that represent a feature or an aspect of an item. In electromechanical product representations, item elements can be used to model ports, interfaces, terminals, or any feature of interest, such as a process variable associated with a signal. The Teamcenter model provides predefined types such as **Network_Port** and **Connection_Terminal** to model functional and physical interfaces. Users can also define their own types of an item element to meet their specific needs.

- Connections

  Connections provide connectivity between a set of item elements or components in a product structure. You use a connection to define logical connectivity between interfaces, physical connectivity between terminals, and functional connectivity between ports. Teamcenter provides two predefined connection types for use in electromechanical product structures:

  - **Network connection** models connectivity between ports of two or more functions.

  - **Connection** models logical or physical connectivity between interfaces of two or more electrical connections or terminals of two or more electrical devices.

- Signals

  Signals are representations of messages and can be instantiated in a product structure. Messages are transmitted in the functional and physical models. Signals can be associated with other electrical product constituents, such as connections of functions, terminals, and connectors, to capture various relationships, such as the transmitter of the signal, the receiver of the signal, and so on. You can also define custom signal types to meet specific needs.

In addition to these abstractions and types, Teamcenter also allows users to enforce restrictions on the usage of these objects in an electromechanical product structure. For example, valid item element types can be tied to a **Connection** type using Teamcenter preferences.

STEP AP212 and KBL models

To support the Mechatronics Process Management design and process within the Teamcenter environment and to share data with other applications, the Teamcenter data model uses industry-standard data models such as STEP AP212, AP210, AP203, AP214, and other wire harness models such as KBL. Teamcenter transfers, stores, and manages all logical, physical, and BOM data in a single secure location. Teamcenter also enables design teams to define wire harnesses consisting of multiple configuration options and variants from a single wire harness design. This section gives an overview of the AP212 and the KBL models.

Standard for the Exchange of Product Model Data (STEP) is an acronym for the Industrial Automation Systems and Integration – Product Data Representation and Exchange international standard (ISO 10303). This standard provides a framework through which industries can exchange and share product information within and between enterprises. It defines standard data exchange protocols that allows engineering data developed under one automated design tool to be read and manipulated by design teams using different automated tools. STEP includes a number of standard application-oriented data models for product description known as application protocols (APs).

STEP AP212 describes the information necessary to customize electrotechnical products, according to the requirements of the installation, and share it between the parties involved in its design, installation, and commissioning. It defines the context, scope, and information requirements for the exchange of design and installation information of electrotechnical equipment and specifies the integrated resources necessary to satisfy those requirements. You can use electrotechnical systems in plants, buildings, or transportation systems such as cars or ships. AP212 does not impose any restriction on the usage of these systems. It covers equipment for power transmission, distribution and generation, electrical machinery, electric light, electric heat, and control and automation systems.

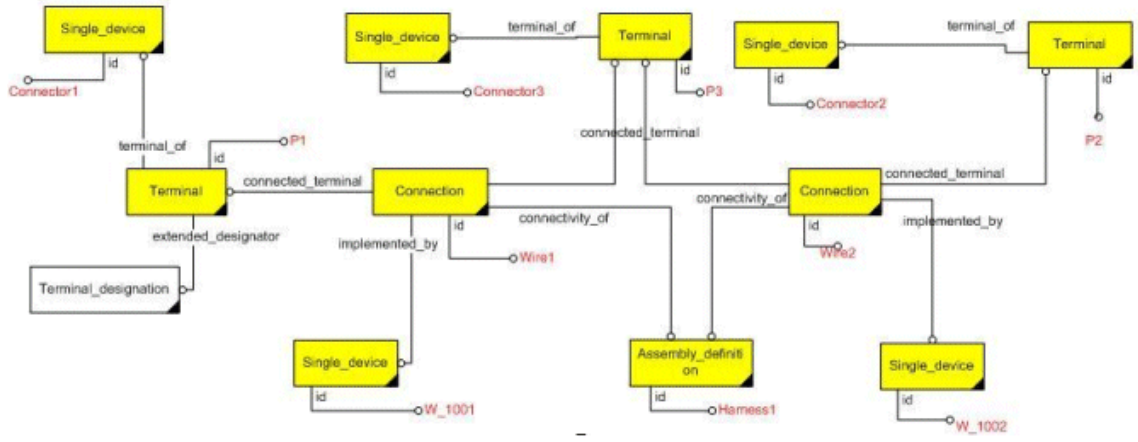
KabelBaumListe (KBL) wire harness description list is a standard for wiring models that enhances the STEP AP212 model. The KBL object model provides high-level components that represent wires,

harnesses, and parts in a vehicle that hosts the harness. This model is a subset of AP212 model and represents wire harness data in a similar way.

The KBL model provides specific object types for representing electrical components including terminals, wires, connectors, and housings. It fully supports the NX wire harness model and allows a wire protection occurrence to be associated to multiple route segments.

The following figure shows an example of electrical connectivity data.

Wire ID	From connector	From pin	To connector	To pin	Part number	Harness
Wire1	Connector1	P1	Connector3	P3	W_1001	Harness1
Wire2	Connector2	P2	Connector3	P3	W_1002	Harness1



KBL functionality types

The following types are provided to support the KBL functionality.

Type	Purpose
HRN_Harness	A harness is an assembly of insulated conductors formed to a predetermined pattern or configuration.
HRN_Module	A module is a physical part of a harness electrically defined by one or more module groups, including the required harness furniture.
HRN_Fixing	A fixing is a component with which the harness is fixed.
HRN_Accessory	An accessory is any supplementary portion of a connector that helps a harness perform its function.

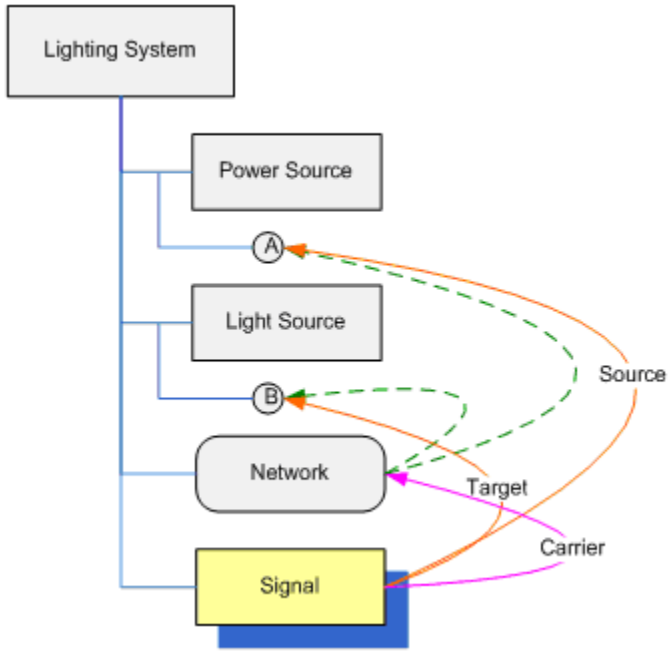
Type	Purpose
HRN_CoPackPart	A part is supplied and installed with the wire harness but without any electrical connection.
HRN_AssemblyPart	An assembly part is a component that contains other subordinate objects.
HRN_WireProtect	A wire protection is a mechanism to describe harness wrappings.
HRN_Cavity	A cavity is a defined space in a housing for the location of an electrical terminal or a cavity plug or seal. It can be empty.
HRN_ConHousing	A con housing is a nonpopulated connector without addressed cavities.
HRN_Slot	A slot is a mechanism to group the cavity objects of a connector housing.
HRN_GeneralWire	A general wire is a physical wire, performing an electrical connection. It may define a single wire or a multicore wire.
HRN_Core	A core is a single conductor of a multicore wire, including its isolation.
HRN_CavityPlug	A cavity plug is a watertight nonelectrical object that fills an empty cavity.
HRN_CavitySeal	A cavity seal is a watertight nonelectrical object that fills a populated cavity.
HRN_GenTerminal	A general terminal object is a device used to terminate a conductor that is usually fixed to a post, stud, chassis, or other conductor to establish an electrical connection.
HRN_Cable	A cable is one or more wires bound together, typically in a common sheath. The individual wires may be covered or insulated.
HRN_Shield	A shield acts as a protection against damage and is mounted on cables and wires.

Wire harness object model

Electrical functional model

In a functional model, electrical components can be functions or functional subsystems.

For example, the following figure shows an automobile lighting system represented as a functional structure in Teamcenter.



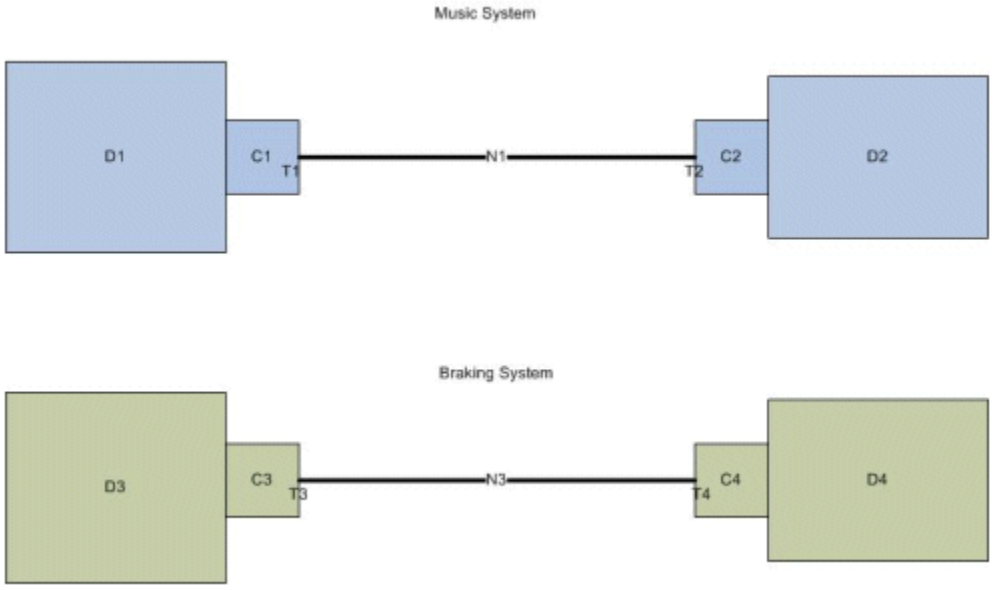
The Teamcenter data model provides the following types of objects to work with the functional models.

- **Functionality** refers to the available functions.
- **Network_Port** refers to the interfaces of functions to the external world.
- **Network** refers to the connectivity element linking various functions through their interfaces.
- **Signal** refers to the message passed between source and target via the carrier.

Electrical logical model

Teamcenter allows you to represent logical electrical connectivity in your product structure. Electrical wiring design typically starts with the definition of logical schematics for each system. A logical schematic consists of connectivity definitions between various electrical devices. It defines the logical connectivity at the system device level, including system level variability. For example, the logical connectivity for the functional subsystems shown in the following figure may consist of a set of electrical devices, connectors, and logical connections between their interfaces.

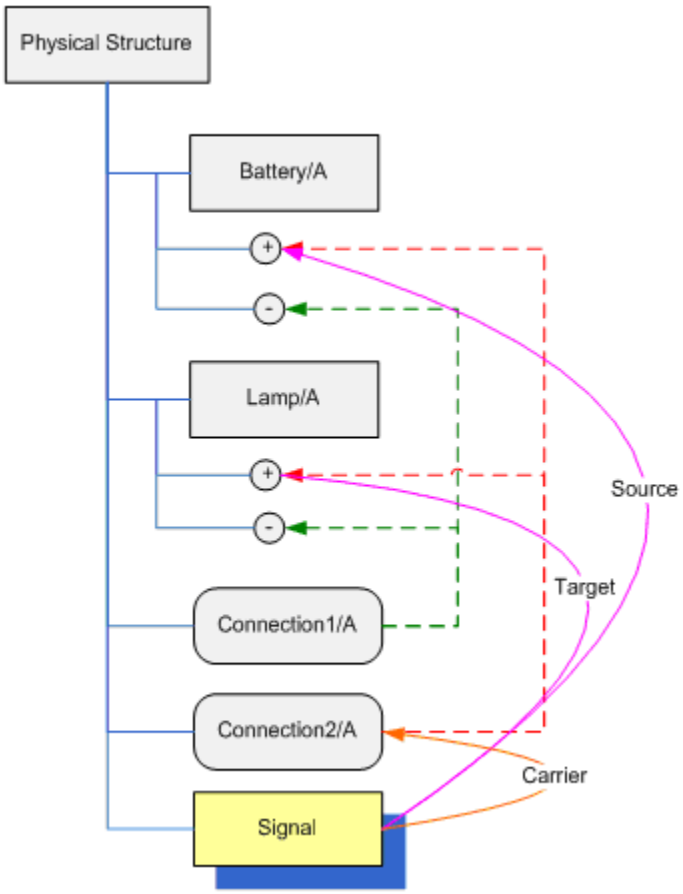
The following figure shows the logical electrical diagram in which D1 and D2 are electrical devices and C1 and C2 are the associated electrical connectors. T1 and T2 are the electrical interfaces of C1 and C2.



To represent electrical connectivity, you can create allocations between the functional model and the logical model.

Electrical physical model

The physical model supports the objects that represent the actual physical objects or devices in a product. For wire harness designs, the physical representation of electrical data consists of wires, harnesses, electrical devices and connectors, and interconnects between them. In Teamcenter, the physical model is represented as a structure of components represented by devices and other physical entities as shown in the following figure.



Wire harness objects

Teamcenter wire harness objects

Teamcenter provides pre-packaged objects that allow the representation of all aspects of an electromechanical product.

Electrical components

Electrical components in the context of wire harness modeling include various electrical devices, connectors, splices, interconnects, wires, cables, and other electrical accessories. In general, you can categorize all these components as parts and model them as *items* in Teamcenter. The Teamcenter data model does not provide any specific item types to represent electrical components but you can define your own item types. You can also use the Classification application to manage these objects.

The KBL harness model defines item types to represent various types of electrical parts, including wires, harnesses, and modules. These item types are not defined by default but can be installed as an option during installation.

In a functional model, electrical components may be *functions* or *functional subsystems*. To manage these objects, the Teamcenter model provides a predefined item type called **Functionality**, a subtype of **Item** that you can use to model the functional breakdown of a product. It also provides the **FunctionalityRevision** item type, which corresponds to a particular revision of a **Functionality** object.

Signals and process variables

Signals represent information or messages transmitted between electrical connectors or devices, for example, an electrical signal passing through the wires joining a battery and a lamp. The Teamcenter implementation of a signal conforms closely to the AP212 definition.

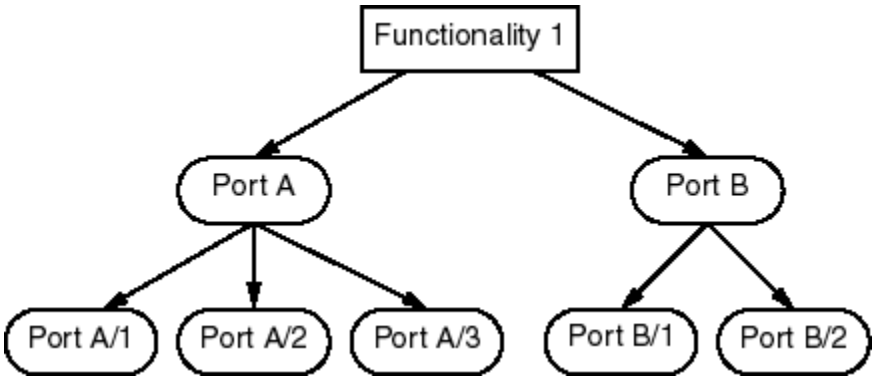
Signals can be instantiated in a product structure and associated with electrical product constituents such as connections, terminals, and connectors to capture various relationships, for example, the transmitter of the signal and the receiver of the signal.

A process variable is a parameter used to control or monitor a process, which may serve as an input, output, or control of the system. Process variable objects are part of the environment. To be processed by the electrotechnical system, process variables must be converted to signals.

Electrical interfaces

Electrical interfaces are exposed by electrical components. For example, the terminals associated with an electrical connector are the interfaces of the connector. The Teamcenter data model provides *item elements* (sometimes called generic design elements, GDEs) to model electrical interfaces. Item elements are subclassed from the **Form** object. You can manage, release, and instantiate item elements in an electrical product structure.

GDEs support decomposition. You can use this capability to define an interface from a hierarchy of other GDEs. An example of an instance diagram of hierarchical ports defined in a function is shown in the following figure.



The following objects allow you to model electrical interfaces.

Object	Purpose
GDE	Models an interface of a product that can be connected, for example, a parallel port on a computer.
GDEOccurrence	Models a usage (an occurrence) of a GDE object in a product structure.
Network_Port	Models interfaces of a product in a functional model and is a subtype of GDE.
Connection_Terminal	Models interfaces of a product in a physical model and is a subtype of GDE.

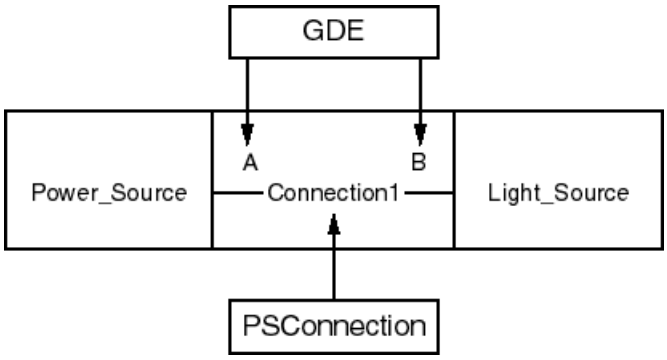
Electrical connections

Electrical connections define general connectivity between a set of item elements or components in a product structure. You can define logical connections, functional networks, and physical connections. The AP212 model separates the connectivity data from the device that implements a connection. The Teamcenter data model allows for separation of connectivity data similar to the AP212 model.

Teamcenter also provides a **TC_Implemented_By** relationship to capture the association between a logical connection and a physical device that implements the connection.

The following objects allow you to model electrical connections.

Object	Purpose
PSConnection	Models connectivity between one or more GDE objects, for example, connectivity between a parallel port of a computer and parallel port of a printer.
Network	Models connectivity in a functional model and is a subtype of **PSConnection**.
Connection	Models connectivity in a physical model and is a subtype of **PSConnection**.
GDELink	Models connectivity between one or more GDE objects. It serves the same functional purpose as a **PSConnection** except that you cannot revise it.

Object	Purpose
	
TC_Link	Models connectivity between one or more GDE objects. It serves the same functional purpose as a **PSConnection** except that you cannot revise it.

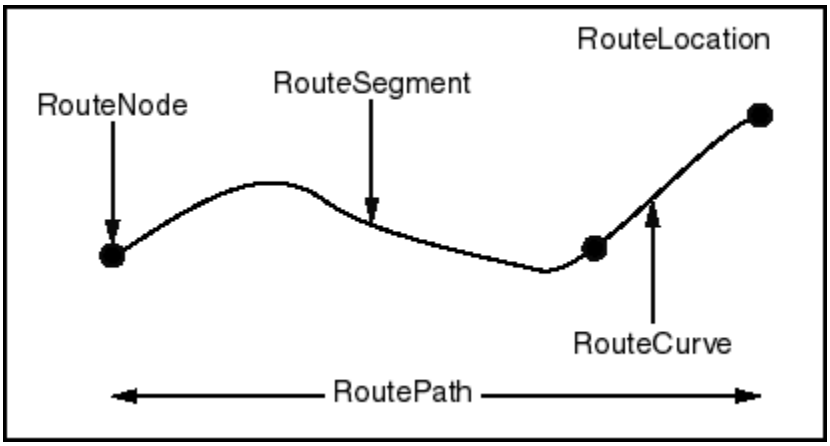
Routes

Routes define the physical route that a wire or cable traverses through a product. The following objects allow you to model routes.

Object	Purpose
RouteNode	Models a point in space.
RouteSegment	Defines the segment of a path with a start node and an end node. You define nodes with **RouteNode** objects. You can optionally define the shape of the segment with a **RouteCurve** object. If you do not define a **RouteCurve** object, Teamcenter assumes the segment shape is a straight line between the start node and the end node.
RouteCurve	Models the 3D shape (**b_spline curve**) associated with a **RoutePath** object or **RouteSegment** object.
RoutePath	Models a 3D or 2D physical path associated with a wire. Define a **RoutePath** object as a set of contiguous **RouteSegment** objects. You can also define it with a set of **RouteNode** objects, in which case an optional **RouteCurve** object defines the shape of the path. When you do not define a **RouteCurve** object, Teamcenter assumes the path consists of straight line segments between **RouteNodes** objects.
RouteLocation	Models a region of space. You can associate objects such as **RouteSegments**, **RouteNodes**, and other electrical items

Object	Purpose
	with a **RouteLocation** object to identify the region of space in which they are located.
RouteLocationRev	Models a revision of a **RouteLocation** object.

The following figure shows how these objects apply to a wire or cable.



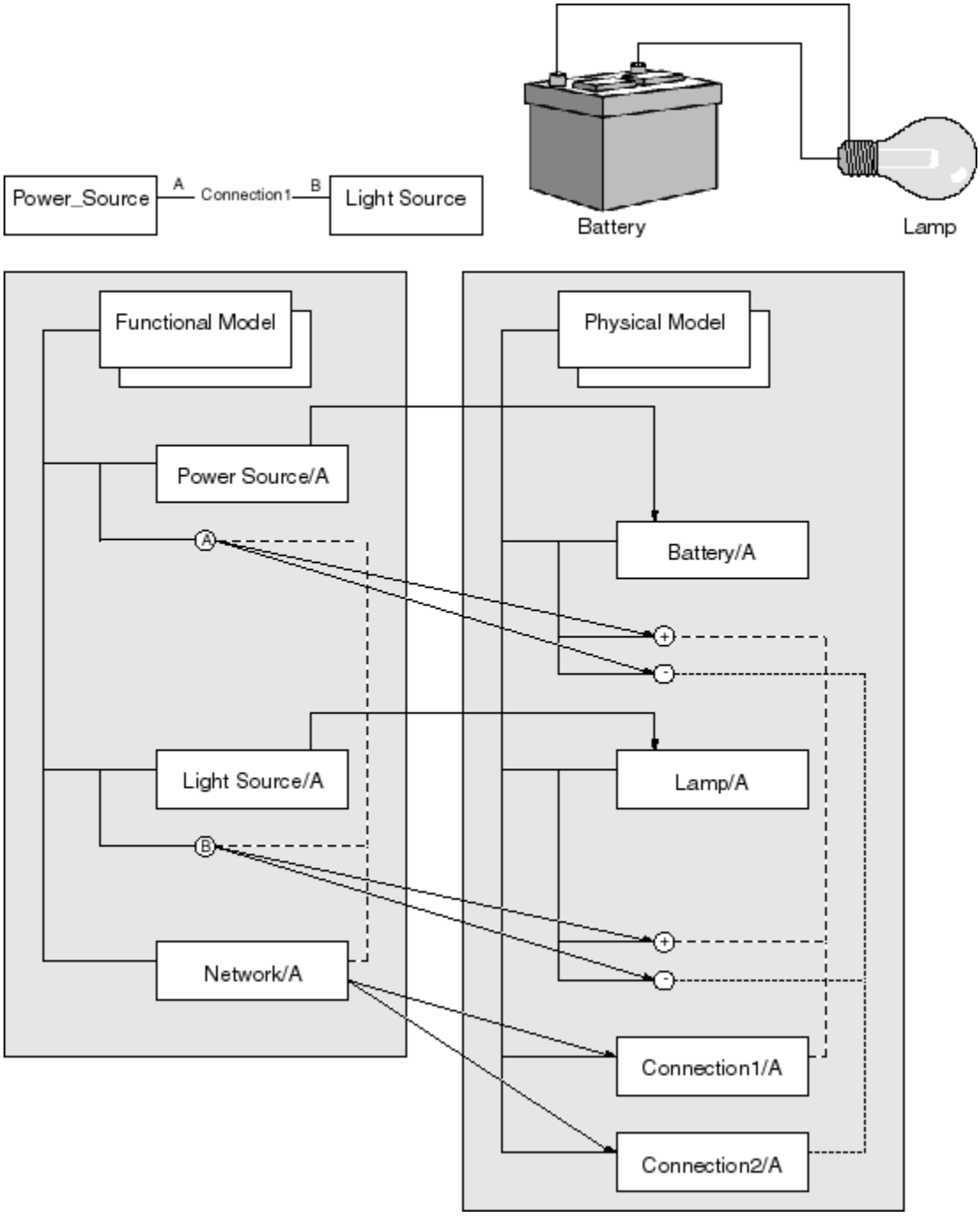
Allocations

Allocations represent mappings from one view in a product structure to another. For example, you can build allocations from functions, networks, and ports in a functional view to devices, connections, and terminals in a physical view. Allocations represent a directional relationship between specific occurrences in two different views. They are independent of the structures they map and are managed independently. You create allocations as a set to map one structure to another; you then define an **AllocationMap** object to group them together. Allocation maps can be revisable and are subclassed from an item. You can also configure allocations independently of the structures they map.

The following objects allow you to model allocations.

Object	Purpose
Allocation	Maps components between different product structures. For example, you can use this object to model the association between components in the functional structure and the physical structure of a product.
AllocationMap	Groups a set of allocations created in the same context.

The following figure shows an example of how to make allocations between the functional model and the physical model.



Relationships

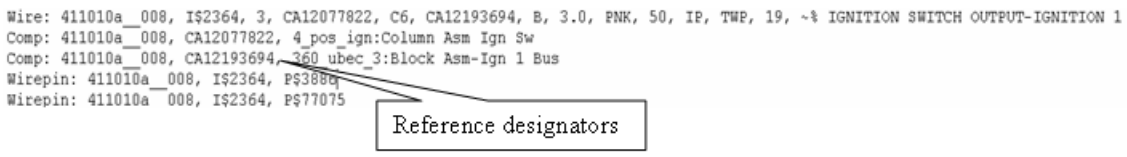
The following relationships allow you to associate fundamental objects.

Relationship	Purpose
Implemented By	Specifies the devices that implement a particular connection.
Realized By	Specifies the interfaces that realize interfaces exposed at a lower level in the assembly structure of a product.
Connected To	Specifies the interfaces connected by a connection.
Routed By	Specifies the route associated with a connection or device.
Device To Connector	Specifies the connectors associated with a device in an electromechanical product.
Assigned Location	Specifies the location (region of space) associated with a routing topology object or product constituent.
Associated System	Specifies the associated system (for example, transmitter or receiver) for a particular signal.
Redundant Signal	Specifies a signal that serves as a redundant signal for another signal.
Process Variable	Specifies the process variable associated with a signal.
SCM_Element_Specification	Associates the Teamcenter SCM object with the binary to establish traceability.

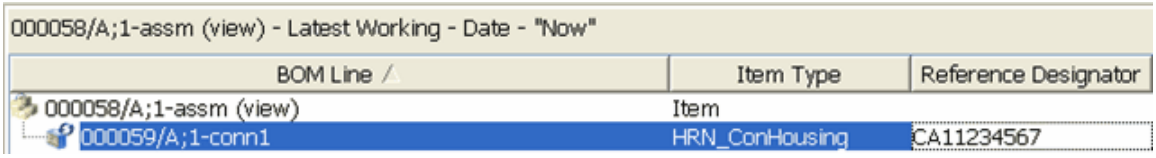
Reference designators

Component reference designators

Reference designators (also called component reference designators) are widely used by electrical and ECAD designers to manage electrical components such as connectors and devices, and their association with physical parts. Reference designators are alphanumeric and are represented graphically on the electrical schematic diagrams. They are used in the electrical interconnects description (net lists) as shown in the following figure.



Teamcenter provides support for reference designators, and they can be specified in any BOM line representing an electrical component as shown in the following figure.



Reference designator information can be exported or imported as occurrence properties using PLM XML. External ECAD applications or the NX routing application can populate and retrieve the reference designator information from the PLM XML file.

Packing and unpacking lines with reference designators

You can pack or unpack product structure lines that include reference designators. For example, if you pack eight occurrences of the same part with different reference designators, the **Reference Designator** property column shows a concatenated set of reference designators, for example, C1, C5-7, C10, and C14-16.

You can configure reference designator packing rules by setting the **BOM_Enable_Ref_Designator_Value_Packing** preference.

> Note:
>
> You cannot edit the reference designators of packed lines.

You can also unpack packed lines that include concatenated reference designators. Each unpacked line shows a single reference designator, for example, C1. Teamcenter validates the correct reference designator format. All reference designators must be in the prefix number format, where prefix is a string of one or more uppercase letters and number is an integer.

To validate the format and uniqueness of reference designators, set the **PS_Reference_Designator_Validation** preference to **true**. This setting also prevents users from editing packed lines. By default, this preference is set to **false** and no validation is performed.

Editing reference designators

Reference designators may be edited only when the product structure is unpacked. Teamcenter verifies that the reference designator is in the correct format at the time of editing. The acceptable format is *Alpha Numeric* or *Alpha Numeric Alpha Numeric*. The alphabetical part of the format should be in uppercase. Examples of the acceptable formats are C1 or SKA21.

To verify the format and uniqueness of reference designators, set the **PS_Reference_Designator_Validation** preference to **true**. By default, this preference is set to **false** and no validation is performed.

Searching for reference designators

You can search for reference designator values for a single given BOM line's immediate children. The search function for BOM lines with specific reference designators is available as a pop-up menu from individual BOM lines. This opens the **Search Reference Designators** window, where you can enter the search parameter. The input for the search should be a text string to be matched against the unpacked value of the property.

Including reference designator in BOM comparison

You can include reference designators in BOM comparisons. You can disable updating of duplicate find numbers of the same item. To do this, set the **PS_Duplicate_FindNo_Update** preference to **disabled**. The **BOMExcludeFromPackCheck** preference allows you to exclude sequence numbers from BOM line packing checks. You can set this preference to **seqno** and BOM lines with distinct sequence numbers can be packed or **none** to exclude them.

3. Allocations

About allocations

In the wire harness design process, the electrical designer creates the functional/logical model and the mechanical designer creates the physical model. The functions in the functional model are represented by some device in the physical model. For example, the power source in a functional model is represented by the battery in the physical model. These representations between items in different models are done using allocations.

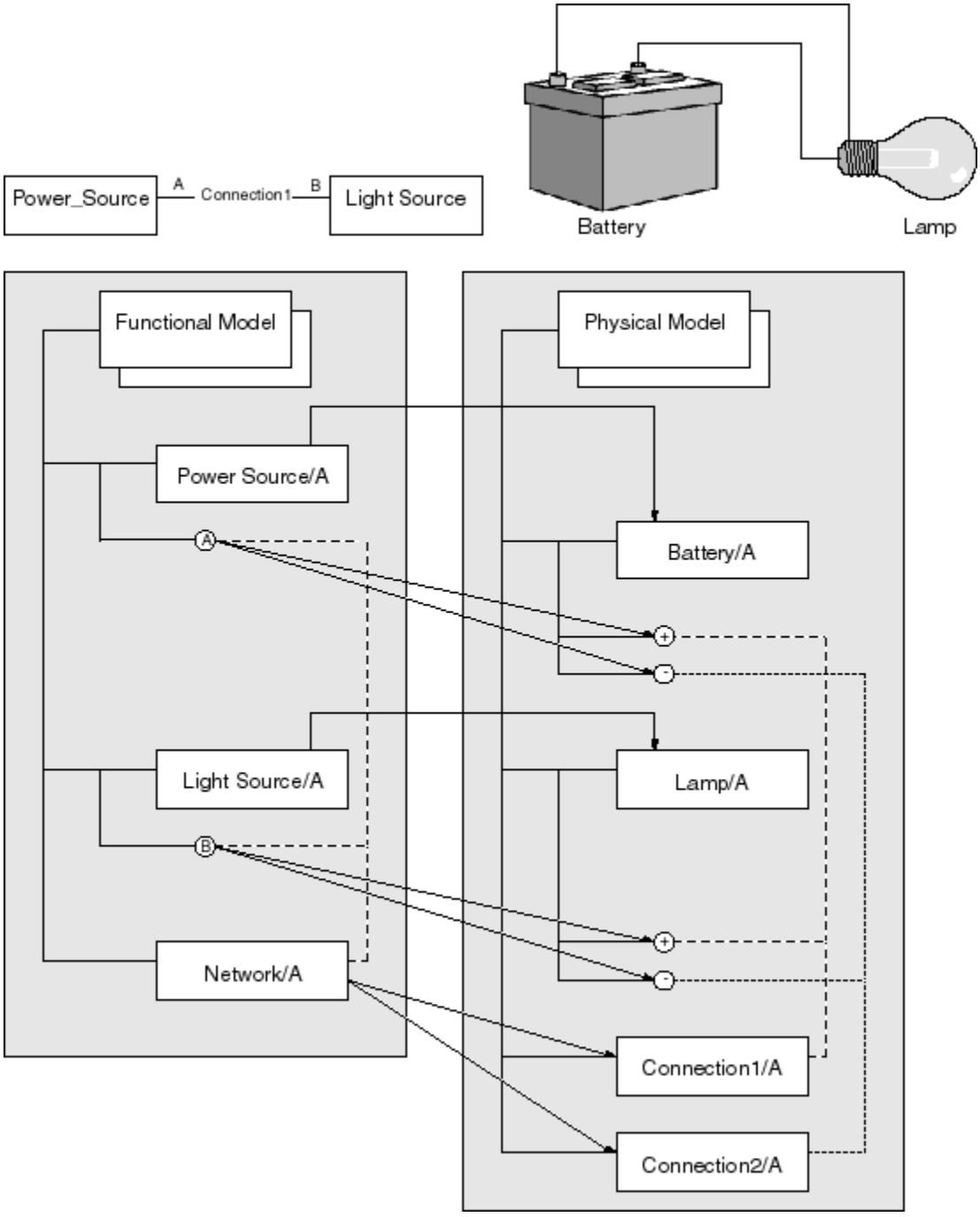
Allocations represent a mapping from one view in a product structure to another view. In AP212, these mappings have a specific meaning. For example, switching from the functional view to the physical view, three types of allocations are represented:

- Port to terminal
- Function to component
- Network to connection

You use allocations to create separate views of a product and then map the views to each other. You can model a purely functional structure that is independent of the parts that eventually realize the functions. With careful analysis, you can use independent groups of parts to design different products with the same functional and logical models.

Allocations allow you to link the actual parameters on physical parts to the requirements on the functional and logical models. This permits you to verify that the components in the product are valid for their proposed purpose.

The following figure shows an example of allocation between the functional model and the physical model.



Allocation properties

You can implement allocations that map between arbitrary views, using component types combinations. For example, you can map a particular function to the set of devices in a physical model that implement it. Similarly, you can map a certain connection (network) in the functional model to the set of connections in the physical model that implement it. This allows you to create a complete general mapping between the two views of a product.

An allocation:

- May be created between two structures of any types, if this action complies with your company's business practices.
- May be created between components of similar types in differing structures. For example, you can create an allocation from a port to a terminal, from a network to a connection, and from a function to a component when linking functional and physical structures.
- Is independent of revisions of the associated components. If you create an allocation between revision A of a component in a functional structure and revision A of a component in a physical structure and subsequently revise the component in the physical structure, then the allocation also exists for the new revision of the component.
- Changes from revision to revision of the associated structures.
- Points from one instance of a component in one structure to a particular instance in a second structure. That is, allocations are made between absolute occurrences of components.
- May be made at various levels in a structure. Allocations need not be made between leaf node components but may be at arbitrary levels in a structure, for example, from a power train electronic control unit (ECU) to an engine ECU.

Working with allocations

Creating allocation maps

Create allocations in a particular context. This context is specified by the top-level components in the structures that are being mapped. To specify the context, define an *allocation map*. An allocation map is an object that specifies how two structures are tied together by a set of allocations that exist between two structures.

Allocations can exist between multiple levels in any structure. However, allocations exist only in the context of a specific product. The product contains multiple representations and provides a context in which to place allocations. The allocation map allows you to specify a set of allocations in a particular product context and (potentially) with a specific purpose.

When you allocate a set of structures, you can define an allocation map that contains all the allocations that have meaning as a set. Defining such an allocation map provides:

- The context for the allocations.
- The organization of the allocations.
- A search mechanism for allocations between specific structures.

- A security mechanism.
- A release mechanism for a group of allocations.
- A mechanism for revising allocation sets.
- A mechanism for building multiple sets of allocations for different purposes between the same set of structures.

An allocation map is a subclass of an item. It has relationships with the structures it maps. It references the BOM views of two items, for example, the functional model of a product and the physical model of a product. All allocations associated with this allocation map must be between absolute occurrences of items in these two specific views.

An allocation map has a *type*. The type specifies the purpose for which a particular allocation map is created, for example, to specify the allocations that define the parts to implement certain functions. It may be created to map the parts consumed by a set of manufacturing operations. An allocation map has a name, allowing you to select different maps for various purposes.

An allocation map has:

- A mapping list, for example, **Source Structure** or **Target Structure**. The source and target views associated with an allocation map apply to all revisions of the allocation map.
- An allocation map name, for example, **2005 Sport Model**.
- An allocation map type, for example, **Function Assignment** or **Consumption**.
- Typed allocations, for example, **implemented in**, **uses**, **consumes**, or **defined by**. You can associate the allocation map to allocation types to suggest that only these associated allocation types are permitted in the allocation map type.

> Note:
>
> Although the allocation map is an item type, do not use an allocation map as a component in a product structure. The allocation map is a hidden object for allocation purposes and is not identified in the source structure or target structure as a component.

Creating allocations

You can create allocations at various levels of a structure, but in each case they specify an object of interest—one that performs a particular function. It is possible that every item revision in a structure could contain an allocation. For example, this may be necessary for manufacturing purposes, where all items must be consumed.

Because allocations are independent of the structures they map, you can create and manage them independently of the structures themselves. For example, a functional designer may design the functional model, while a mechanical engineer designs the physical model. When both structures are complete, a systems engineer may map the functional model into the physical model.

> Note:
>
> The model is the representation of a particular aspect of the product, not the CAD model. For example, the functional model of a car represents a product containing an air conditioning system, power braking, power steering, and other functionality. In Teamcenter, the model is represented by the product structure.

Because you can configure structures in Teamcenter, you can create allocations between variable product structures. Allocations are valid only in a particular context, and therefore all allocations between two structures are part of a group and Teamcenter evaluates them as a unit. The allocations tie together various components and revisions of items in the structures.

Allocation have a direction—each allocation must have a source and a target. The direction shows the driver (source) of the relationship, for example, **Function 4 is allocated to ECU 1**.

When you create an allocation, Teamcenter automatically assigns it a unique identifier. The identification scheme is determined by user exits set by the Teamcenter administrator.

Optionally, a customizer can create subtypes of allocations and allocation maps using the Business Modeler IDE. The Teamcenter administrator can set preferences that determine the allocation subtypes that are valid in a particular allocation map type and, similarly, the allocation maps that are valid between structure type (BOM view types). The administrator also determines whether each allocation subtype is valid for:

- One source and one target.
- One source and multiple targets.
- Multiple sources and one target.
- Multiple sources and multiple targets.

4. Configuring and administering Wiring Harness Design Tools Integration

Configuring and administering Wiring Harness Design Tools Integration

You can configure and administer the Wiring Harness Design Tools Integration functionality to suit your site requirements.

Administering PLM XML data transfers

The PLM XML export import functionality in Teamcenter allows you to export objects to external systems and import objects from external systems. A PLM XML file is created as the mechanism for exchanging data between Teamcenter and the external application. It supports the transfer of objects such as wire harness data, items, datasets, BOMs, forms, folders, and system data (for example, business rules, organization data, type definitions, and saved queries). The PLM XML schema is extended to support the wire harness integration object model.

Transfer mode objects define the import or export context by applying closure (traversal) rules, filter rules, and property set rules to the input or output data. These rules may be stored in the database as a static set or they may be applied only to a specific session. It is important to use the correct transfer mode with PLM XML export and import operations in order to accomplish the data transfer of desired entities.

Use the **MechatronicsFoundationDataExport** transfer mode to export ECAD data from Teamcenter. This transfer mode can be used to export core data such as connections and signals.

Use the **HRNExchange** transfer mode to export wire harness data from Teamcenter.

Use the **IncrementalImport** transfer mode to import ECAD data into Teamcenter.

When you select the **MechatronicsFoundationDataExport** transfer mode, the following objects and information are transferred:

- Signals
- Associated systems of a signal object (source, target, transmitter)
- Connections
- Connection information
- **ImplementedBy** information

- **RealizedBy** information
- Interfaces
- Allocations
- Allocation maps
- Routes
- Nodes
- Segments
- Route location
- Process variables
- Signal values
- Various KBL types

To export Embedded Software Solutions related objects and relations, add these closure rules:

- `TYPE BOMLine TYPE * PROPERTY bl_embeds_lines_tags PROCESS+TRAVERSE`
- `TYPE BOMLine TYPE * PROPERTY bl_dependentOn_lines_tags PROCESS+TRAVERSE`
- `TYPE BOMLine TYPE * PROPERTY bl_gatewayOf_lines_tags PROCESS+TRAVERSE`

Export electromechanical structure

To export an electromechanical structure do one of the following:

- Run the **plmxml_export** command line utility, as described in the *Teamcenter Utilities*.
- Choose the **Tools→Export** menu command in My Teamcenter.
- Choose the **Tools→Export** menu command in Structure Manager.

Mapping data for PLM XML import and export transfers

The following table lists the equivalent data type mappings among Teamcenter, PLM XML, and the AP212 model.

Teamcenter	AP212	PLM XML
Item	**Item**	**Product**
ItemRevision	**Item_version**	**ProductRevision**
BOMViewRevision	**DDID**	**ProductRevisionView**
BOMViewRevision	**Assembly_definition**	**ProductRevisionView**
PSOccurrence	**Single_device**	**ProductInstance**
PSOccurrence	**Next_higher_assembly**	**ProductInstance**
GDE type	**Interface_terminal**	**TerminalInstance** or **Occurrence**
Connection	**Connection**	**FlowConnection**, **FlowConnectionRevision**, **FlowConnectionRevisionView**, **ConnectionInstance**
Connected_to (property)	**Connectivity_definition_relationship**	**GDEReference**
Signal	**Signal**	**Signal** **SignalRevision** **SignalInstance**
Signal_value (property)	**Signal_value**	**SignalValue**
GDE type	**Process_variable**	**ProcessVariable**
RouteNode	**Node**	**RouteNode**
RoutePath	**Route**	**Route**
RouteSegment	**Section**	**RouteSection**
RouteNode	**Section_interface**	**RouteNodes**
RouteNode	**Section_end**	**RouteNode**
RoutePath	**Path**	**Route**
RouteNode	**Path_node**	**RouteNode**
RouteSegment	**Path_segment**	**RouteSection**
RoutedBy (property)	**Routed_object**	**RouteData/GDEReference**
RoutePath	**Routed_segment**	**RouteDataSegment**
RouteCurve	**Curve_3d**	**BSplineCurve**

Teamcenter	AP212	PLM XML
Property	**Predefined_data_element**	**DataElement**
Form Attribute	**Material**	**MaterialName**
None	**Body_breadth**	**Ylength**
None	**Body_height**	**Zlength**
Real_length	**Body_length**	**Xlength**
Cross_section (property)	**Cross_section**	**CrossSectionalArea**
Form Attribute	**Component_colour**	**ColourName**
Form Attribute	**Mass**	**MassProperties**
Form Attribute	**Outside_diameter**	**OutsideDiameter**
Form Attribute	**Rated_current**	**RatedCurrent**
Form Attribute	**Rated_power**	**RatedPower**
Form Attribute	**Rated_voltage**	**RatedVoltage**
Form Attribute	**Storage_temperature**	**StorageTemperature**
Form Attribute	**Value_with_unit**	**ValueWithUnit**
ReleaseStatus	**Approval_status**	status attribute on **Approval**
Effectivity	**Effectivity**	**Effectivity**

Data mapping for KBL PLM XML import and export transfers

The equivalent data type mappings between Teamcenter PLM XML and the KBL model are described here.

Data type	Description
Functionality	An action or behavior by which an electrical product fulfills its purpose. Functionality is the primary class representing a functionality in the functional model. PLM XML example for a **Functionality** item: `<Product id="id19" name="Switch" accessRefs="#id9" subType="Functionality" productId="000007">` `...` `</Product>` PLM XML example for a **Functionality** item revision:

Data type	Description
	<ProductRevision id="id7" name="Switch" accessRefs="#id9" subType="FunctionalityRevision" masterRef="#id19" revision="A"> ... </ProductRevision>
Port	Occurrences of interface ports used to represent the access point of a functional module in a given context, for example, the USB 2.0 port on a computer processor. A port may be a network port or a connection terminal. A network port is a port in a functional model. A PLM XML example for a network port follows: <Terminal id="id21" name="Switch_port" accessRefs="#id10" subType="Network_Port"> ... </Terminal> In Structure Manager, a network port is represented by **GDEInstance** for the sequence number and **OccurrenceId** for the transformation as follows: <GDEInstance id="id31" name="Switch_port" partRef="#id32" instancedRef="#id21" quantity="1" instanceNumber="1">... </GDEInstance> <Occurrence id="id29" instancedRef="#id21" parentRef="#id5" instanceRefs="#id31">... </Occurrence> All integrating applications must use unique version and label attributes to avoid duplicate ports, as in the example: <GDEInstance id="id31" name="p1" partRef="#id32" instancedRef="#id14" quantity="1" instanceNumber="1"> <Applicationref version=”p2007140618a1025954372” application=”Teamcenter” label=”ylOhnMYETLWLEB”> </ApplicationRef> </GDEInstance> A connection terminal is a port in a physical model. The PLM XML element is the same as a network port but with a **Connection_Terminal** subtype, as follows: <Terminal id="id21" name="Switch_port" accessRefs="#id10" subType="Connection_Terminal">... </Terminal>
Revisable Connection	An object that defines the connectivity between two or more terminals and may be revised. A revisable connection may be a network or a connection. A network is a revisable connection in a functional model. A PLM XML example for a network is as follows: <FlowConnection id="id34" name="network" accessRefs="#id10" subType="Network" catalogueId="000013">

Data type	Description

```
...
</FlowConnection>
```

A PLM XML example for a network revision is as follows:

```
<FlowConnectionRevision id="id31"
name="network" accessRefs="#id10" subType="NetworkRevision"
masterRef="#id34" revision="A">
...
</FlowConnectionRevision>
```

A connection is a connection in a physical model. The PLM XML element for a connection is the same as for network with subtypes for connection and connection revisions. A PLM XML example for connection is as follows:

```
<FlowConnection id="id34" name="network"
accessRefs="#id10" subType="Connection" catalogueId="000013">
<ApplicationRef version="RqMJuT3oxp5BdC" application="TcEng"
   label="RqMJuT3oxp5BdC”/>
</FlowConnection>
```

A PLM XML example for a connection revision is as follows:

```
<FlowConnectionRevision id="id31" name="
network" accessRefs="#id10" subType="ConnectionRevision"
masterRef="#id34" revision="A">
...
</FlowConnectionRevision>
```

Non Revisable Connection (TC_Link)

An object that defines the connectivity between two or more terminals that cannot be revised. A PLM XML example for a **TC_Link** is as follows:

```
<Link id="id21" name="NR_conn"
accessRefs="#id10" subType="TC_Link">
...
</Link>
```

Signal

A physical representation of messages processed in a system. A PLM XML example for a signal is as follows:

```
<Signal id="id2" name="signal1" accessRefs="#id3"
subType="Signal" catalogueId="000018">
...
</Signal>
```

A PLM XML example for a signal revision is as follows:

```
<SignalRevision id="id2" name="signal1"
accessRefs="#id3" subType="SignalRevision" masterRef="#id14"
revision="A">
```

Data type	Description
	`` ... </SignalRevision> ```
Process Variable	Represents variables in the system that must be processed and controlled. A PLM XML example for a process variable is as follows: ``` <GDE id="id12" name="temperature" accessRefs="#id3" subType="ProcessVariable"> ... </GDE> ```
Allocations	Represent a mapping from one view in a product structure to another view. A PLM XML example for an allocation is as follows: ``` <Allocation id="ALL_id37" subType="Allocation" relatedRefs="#OCC_id28 #ER_id31"> <Reason value="functionalToPhysical"/> </Allocation> ``` PLM XML examples for an allocation map and allocation map revision is as follows: ``` <AllocationGroup id="AG_id36" subType="ProductRevisionView" targetRef="#ER_id33" sourceRef="#PRV_id25"/> <AllocationGroupRevision id="AGR_id35" masterRef="#AG_id36" memberRefs="#ALL_id37" revision="A"/> ```
Route	A route is the topology and path associated with a wire or a connection. A route is represented by the **RouteNode**, **RouteSegment**, **RouteCurve**, **RoutePath**, and **RouteLocation** objects. A PLM XML example for a route node is as follows: ``` <RouteNode id="id48" position="-24 -10.191 10.224"/> ``` A PLM XML example for a route segment is as follows: ``` <RouteDataSegment id="id47" courseRefs="#id55"/> <Route id="id55" subType="RoutePath" courseRefs="#id48 #id49 #id50 #id51 #id50 #id51 #id52 #id53 #id54"/> ``` A PLM XML example for a route path is as follows: ``` <Route id="id55" subType="RoutePath" courseRefs="#id48 #id49 #id50 #id51 #id50 #id51 #id52 #id53 #id54"/> ``` A PLM XML example for a route location is as follows: ``` <RouteData id="id46"> ..<RouteDataSegment id="id47" courseRefs="#id55"/> </RouteData> <RouteNode id="id48" position="-24 -10.191 10.224"/> <RouteSection id="id49"/> ```

Data type	Description

```
<RouteNode id="id50" position="-0.692 -0.712 6.457"/>
<RouteSection id="id51"/>
<RouteNode id="id52" position="-0.003 -0.003 2"/>
<RouteSection id="id53"/>
<RouteNode id="id54" position="0.034 -1.531 -4"/>
<Route id="id55" subType="RoutePath" courseRefs="#id48 #id49 #id50
 #id51 #id50 #id51 #id52 #id53 #id54"/>
```

Relationships

Teamcenter allows the transfer of **Implemented By**, **Realized By**, **Associated System**, **Redundant Signal**, and **Process Variable** relationships.

A PLM XML example for an **Implemented By** relationship is as follows:

```
<Occurrence id="id69" instancedRef="#id71"
parentRef="#id5" instanceRefs="#id75">
  <Reference id="id82" occurrenceRef="#id80" type="implementation"/>
</Occurrence>
```

A PLM XML example for a **Realized By** relationship is as follows:

```
<Occurrence id="id47" instancedRef="#id40"
parentRef="#id32" instanceRefs="#id50" sourceRef="#id49">
  <Reference id="id55" occurrenceRef="#id53" type="realisation"></
Reference>
</Occurrence>
```

A PLM XML example for an **Associated System** relationship is as follows:

```
<Occurrence id="id47" instancedRef="#id49"
   parentRef="#id5" instanceRefs="#id53">
 <Reference id="id58" occurrenceRef="#id56" type="signalSource">
   /** Signal Source */
  <ApplicationRef version="BqLJPoGcwVL3RB" application="TcEng"
   label="BqLJPoGcwVL3RB"/>
 </Reference>
 <Reference id="id65" occurrenceRef="#id63" type="signalTarget">
   /* Signal Target */
 <ApplicationRef version="RqHJPoGcwVL3RB" application="TcEng"
   label="RqHJPoGcwVL3RB"/>
 </Reference>
 <Reference id="id69" occurrenceRef="#id35"
type="signalTransmitter">
        /* Signal Transmitter */
 <ApplicationRef version="RyJJPoGcwVL3RB" application="TcEng"
   label="RyJJPoGcwVL3RB"/>
 </Reference>
</Occurrence>
```

A PLM XML example for a **Redundant Signal** relationship is as follows:

```
<Occurrence id="id47" instancedRef="#id49"
   parentRef="#id5" instanceRefs="#id53">
  <Reference id="id80" occurrenceRef="#id78" type="redundancy">
   <ApplicationRef version="RiDJPoGcwVL3RB" application="TcEng"
   label="RiDJPoGcwVL3RB"/>
```

Data type	Description
	</Reference> </Occurrence> A PLM XML example for a **Process Variable** relationship is as follows: <Occurrence id="id47" instancedRef="#id49" parentRef="#id5" instanceRefs="#id53"> <Reference id="id73" occurrenceRef="#id71" type="processVariable"> <ApplicationRef version="RaEJPoGcwVL3RB" application="TcEng" label="RaEJPoGcwVL3RB"/> </Reference> </Occurrence>
Reference designator	Teamcenter allows transfer of reference designators of ECAD components using PLM XML. To transfer reference designators, add the **forHRNExchange** rule to the property set of the export transfer mode: **Type | BOMLine | PROPERTY | bl_ref_designator | DO** A PLM XML example containing reference designator information for a connector occurrence is as follows: <Occurence id="id30" instancedRef="#id33" sourceRef="#id32" instanceRefs="#id42" parentRef="#id7"> <UserData id="id29"> <UserValue value="CA11234567" title="bl_ref_designator"> </UserValue> </UserData> </Occurrence>

Examples of PLM XML for KBL elements

Teamcenter uses the **forHRNExchange** transfer mode for the import and export of KBL related types, relations, and attributes. The following PLM XML elements represent KBL data:

- **<HarnessProduct>**

  This is a subclass of **Product** and corresponds to a product related to an electrical harness definition, as defined in the KBL standard. All KBL item types are represented as **<HarnessProduct>** in PLM XML, for example:

```
<HarnessProduct id="id26" name="Harness_name" subType="HRN_Harness"
productId="000025" type="harness" harnessProductType="harnessPartType">
```

Attribute	Description
ProductId	Item ID
Name	Name of the item
subType	Item type

Attribute	Description
Type	Specifies the type of product. The **type** attribute must be one of the following types: **harness** **module** **connectorHousing** **accessory** **connectorCavityPlug** **connectorCavitySeal** **terminal** **wire** **wireProtection** **fixing** **general** **composite**
harnessProductType	A string describing the type of product. In Teamcenter, this is represented by the **part_type** attribute on the item master form or property.
code	A code for the product. Only relevant when the type is **connectorHousing**, for example: `<HarnessProduct id="id58" name="ConnHouse_name" subType="HRN_ConHousing" productId="000286" type="connectorHousing" code="housing_code_for_chousing">`

- **<WireCore>**

  This represents a single wire in a multicore cable. In Teamcenter, this corresponds to the **HRN_Core** GDE type, for example:

  ```
  <WireCore id="id22" name="Core_1_name"
  subType="HRN_Core" wireCoreType="core_part_type">
  ```

Attribute	Description
Name	Name of the GDE object.
subType	GDE type (**HRN_core**).
wireCoreType	A string describing the type of wire core. In Teamcenter, this is represented by the **part_type** attribute on the **HRN_Core** GDE object property.

- **<ConnectorCavity>**

This represents a cavity in a connector into which a wire may be fixed. In Teamcenter, this corresponds to the **HRN_Cavity** GDE type, for example:

```
<ConnectorCavity id="id45" name="Cavity_1_name"
subType="HRN_Cavity" connectorCavityType="cavity_part_type">
```

Attribute	Description
Name	Name of the GDE object
subType	GDE type (**HRN_Cavity**)
connectorCavityType	A string describing the type of connector cavity. In Teamcenter, this is represented by the **part_type** attribute on the **HRN_Cavity** GDE object property.

- **<ConnectorCavityGroup>**

  This represents a group of cavity connectors represented in Teamcenter by **HRN_Slot**, for example:

  ```
  <ConnectorCavityGroup id="id46" name="Slot_2_name"
  subType="HRN_Slot" connectorCavityGroupType="slot_part_type">
  ```

Attribute	Description
Name	Name of the GDE object.
subType	GDE type (**HRN_Slot**).
connectorCavityType	A string describing the type of connector cavity group. In Teamcenter, this is represented by the **part_type** attribute on the **HRN_Slot** GDE object property.

- **<WireProtectionAreaRelation>**

  This relates an occurrence of a **WireProtection** element to a **RouteSection** element that specifies where the protection applies.

  In addition to specifying the related objects, this element holds the data for the protection area, including end location, start location, gradient, and taping direction.

  In Teamcenter, this corresponds to the **HRN_protection_area** relation that relates a wire protection object to a segment. The protection area data is attached to the wire protection occurrence by the **HRN_set_protection_area_data** relation.

  Example:

  ```
  <WireProtectionAreaRelation id="id276" relatedRefs="#id256 #id272"
  gradient="44" endT="40" startT="32.4" tapingDirection="right"/>
  ```

Attribute	Description
relatedRefs	References the following elements in the order listed: **<Occurrence>** Wire protection occurrence in the relation. **<RouteSection>** Segment object for which the protection area is defined.
endT	End location
start	Start location
Gradient	Gradient
tapingDirection	Takes the enum value, **right** or **left**. This corresponds to the **taping_direction** string attribute in Teamcenter.

- **<RouteSectionAssignment>**

  This relation relates an occurrence to a **RouteSection** element on which it lies. For example, the occurrence may be a fixing on a wire. In addition to specifying the related objects, this element holds the data for the fixing assignment including location and orientation value.

  In Teamcenter, this corresponds to the **HRN_fixing_assignment** relation that relates a fixing or accessory object to a segment. The fixing assignment data is attached to the occurrence using the **HRN_set_fixing_assignment_data** relation.

  Example:

  ```
  <RouteSectionAssignment id="id193" relatedRefs="#id173 #id189" zAxis="0 0 0" t="0"/>
  ```

Attribute	Description
relatedRefs	References the following elements in the order listed: **<Occurrence>** Fixing or accessory occurrence or their subtypes in the relation. **<RouteSection>**

Attribute	Description
	Segment object for which the assignment is defined.
T	Location value
xAxis	Orientation value (**x=, y=, x=**)

- **<RouteNodeAssignment>**

  This relation relates an occurrence to a **RouteNode** element on which it lies. The occurrence is the referenced component at the node.

  In Teamcenter, this corresponds to the **HRN_referenced_component** relation that relates a part occurrence to a **routeNode** object.

  For example:

```
<RouteNodeAssignment id="id59" relatedRefs="#id998 #id58"/>
```

Attribute	Description
relatedRefs	References the following elements in the order listed: **<Occurrence>** Occurrence in the relation. **<RouteNode>** Node object for which the assignment is defined.

- **<Reference type= “associated”>**

  The **HRN_Cavity** type or subtype may be related to occurrences of the **HRN_CavitySeal**, **HRN_CavityPlug**, or **HRN_GenTerminal** type or subtype.

  This relation is represented by the **<Reference>** attribute having **type= “associated”** on the **HRN_Cavity** occurrence element. In Teamcenter, the **HRN_associated_part** relation corresponds to this reference attribute with **type=”associated”**.

  For example:

```
<Occurrence id="id51" instancedRef="#id40" instanceRefs="#id54" sourceRef="#id53">
   <Reference id="id59" occurrenceRef="#id57" type="associated"></Reference>
   <Reference id="id94" occurrenceRef="#id92" type="associated"></Reference>
   <Reference id="id110" occurrenceRef="#id108" type="associated"></Reference>
</Occurrence>
```

Attribute	Description
occurrenceRefs	References the occurrence that participates as the secondary occurrence of the **associated_part** relation.
Type	For the **associated_part** relation, the type value should always be **associated**.

Property elements in KBL PLM XML

The attributes defined on the master form and occurrence attributes of KBL types are represented as **PropertyGroup** and **Userdata** in PLM XML. The various property elements in PLM XML are as follows:

Property elements	Description
<Color>	Represents color attributes for the KBL types, for example: `<Colour id="id249" type="outside_colour" colourId="yellow"/>`
<BendRadius>	Represents the bend radius attribute for the KBL types, for example: `<BendRadius id="id167" value="32.33"/>`
<OutsideDiameter>	Represents the outside diameter attribute for the KBL types, for example: `<OutsideDiameter id="id166"><ValueWithUnit id="id165" value="87.56"/></OutsideDiameter>`
<WireGauge>	Represents gauge attributes for objects of the **HRN_GeneralWire** and **HRN_Core** types and subtypes, for example: `<WireGauge id="id177" value="21.34"/>`
<CrossSectionalArea>	Represents the cross-sectional area attribute for the KBL types. `<CrossSectionalArea id="id164"><ValueWithUnit id="id163" value="99.99000000000001"/></CrossSectionalArea>`
<CompatibleWireSize>	Represents the **wire_size** attribute on objects of the **HRN_CavitySeal** types and subtypes, for example:

Property elements	Description
	<CompatibleWireSize id="id248"> ...<ValueWithUnit id="id247" value="99.89"/> </CompatibleWireSize>
<MaterialSpecification>	Specifies the material reference system and material ID attribute of all KBL part revisions, for example: <MaterialSpecification id="id183" materialId="M-ID-1120" referenceSystem="Sys-1000" /> It also represents the **plating_material** attribute on objects of the **HRN_GenTerminal** types and subtypes, for example: <MaterialSpecification id="id182" usage="plat_mat_genterm"/>
<MassProperty>	Specifies the **mass_information** attribute of all KBL part revisions, for example: <MassProperty id="id184"> <ValueWithUnit id="id163" value="99.99000000000001"/> </MassProperty>
<Copyright>	Represents the copyright note information for the KBL types. In Teamcenter, the copyright note is a form associated with the KBL item or GDE type or subtype object. For example: <Copyright id="id199" value="copyright_info_goes_here"/>
<LengthProperty>	Occurrences may have **length_type** and **length_value** specified on them. In Teamcenter, the **HRN_set_wire_length_data** ITK function associates the **length_type** and **length_value** pair with the occurrence. This is represented by the **LengthProperty** property element in PLM XML.

UserData in KBL PLM XML

The following KBL type attributes are represented as **UserData** in PLM XML.

Attribute	Properties on master form of type or subtype
Process_type	Master form properties on all KBL types.

Attribute	Properties on master form of type or subtype
Process_value	
Car_class_level2	**HRN_Harness** and **HRN_Module**
Car_class_level3	
Car_class_level4	
Company_name	
Model_year	
Type_depen_param	**HRN_WireProtect**
Min_length	**HRN_GeneralWire** and **HRN_Core**
Max_length	
Wirespec	
Width	
Height	
Cover_thickness	
Cable_designator	

The occurrence-specific attributes for the following KBL types are represented as **UserData**:

- The **Connector Housing** occurrence or subtype may have usage defined as an occurrence attribute. This is represented as follows:

```
<Occurrence id="id46" instancedRef="#id51"
occurrenceRefs="id65" parentRef="#id5" instanceRefs="#id62"
occurrenceId="ConnId1" sourceRef="#id61">
   <UserData id="id48" type="ConnectorUsage">
    <UserValue value="usage" title="usage"/>
   </UserData>
</Occurrence>
```

- The **Cavity** occurrence or subtype may have **position_on_wire** specified as an occurrence-specific attribute. This is represented as follows:

```
<Occurrence id="id65" instancedRef="#id57"
parentRef="#id46" instanceRefs="#id69" occurrenceId="CavId_1"
sourceRef="#id68">
   <UserData id="id67" type="CavityPositionOnWire">
    <UserValue type="real" value="40" title="posi_on_wire"/>
   </UserData>
</Occurrence>
```

- The **GeneralWire/Core** occurrence or subtypes of these types may have **position_on_wire** specified as an occurrence-specific attribute. This is represented as follows:

```
<Occurrence id="id67" instancedRef="#id59"
parentRef="#id48" instanceRefs="#id71" occurrenceId="genwire_1"
sourceRef="#id73">
   <UserData id="id67" type="Multiplier">
    <UserValue value="multiplier_val" title="multiplier"/>
   </UserData>
   <UserData id="id67" type="Offset">
    <UserValue value="offset_val" title="offset"/>
   </UserData>
   <UserData id="id67" type="SeperationCode">
    <UserValue value="separation_code_val" title="separation_code"/>
   </UserData>
</Occurrence>
```

Configuring for MCAD

Configure system for information exchange

Before exchanging information between EDA, MCAD, and Teamcenter, you must configure your system.

1. Create an application interface business object.

2. Configure **Application Ref** for export.

Creating an application interface business object

Use the default **AppInterface** type **NXRout_AIType** for MCAD data exchange. If you prefer to use the custom export/import transfer modes for MCAD data exchange, you can do so by creating an **AppInterface** type object and setting the preference value **NXRouting_Electrical_AI_Type** to **AppInterface** type object. The default value of this preference is set to **NXRout_AIType**.

Configure Application Ref for export

Get the **NX Electrical Application Ref** output during export of PLM XML file.

1. Log on as an administrator and select PLM XML/TC XML Export Import Administration.

2. Select the export transfer mode that is associated with MCAD data exchange.

3. Click **+** to add the following data:

```
CLASS | * | CLASS | PLMAppUID | PROPERTY | application_uid |

TRAVERSE_AND_PROCESS
```

Working with sample SOA wire harness structure

About sample SOA wire harness structure

Teamcenter comes bundled with a sample SOA wire harness structure client in C++ and Java that depicts a typical harness design exchange with variants and options. You can execute and use this sample client as a guide to create your own client integrations with Teamcenter. The sample client depicts how you can:

- Create maximum complexity harness structure with options and variants. You can see how a harness structure containing devices, connectors with pins, connections, and wires is created. This will take you through the process of populating topology information such as routes with segments, curves and nodes, and creating associations between routes and wires. You can see how the option set is created when variants are applied on connectors, wires, and connections. It depicts how a saved variant configuration is created for every possible option.

- List the saved variant configurations of maximum complexity and get the configured components for each configuration.

- Update the maximum complexity structure components along with variants and options. This includes how you can delete and/or add wires, connectors, connections, and saved options.

- Create allocations between components of electrical harness structure and regular product structure. The process also shows how to delete and edit allocations between connectors.

C++ sample client

C++ sample SOA wire harness client

The C++ sample client is located under the *soa_client***cpp\samples\WireHarness** directory, installed as part of your Teamcenter kit. You can build and execute this project from either Visual Studio 2008 or the command prompt. Before running the sample client, ensure that you have installed Wiring Harness Design Tools Integration. For a four-tier Teamcenter installation, ensure that you have Teamcenter Web tier server and Pool Manager installed. For a two-tier Teamcenter installation, ensure that you have TcServer running.

Execute the client using Visual Studio

1. Add the location of the Teamcenter Services library to the PATH environment variable:

   *soa_client***cpp\libs\wnti32;%PATH%**, where *soa_client* is the location of the root folder from the Teamcenter CD-ROM.

2. Load the project.

3. Choose **File→Open→Project** and browse to *soa_client***cpp\samples\WireHarness\WireHarness.vcproj**.

4. Choose **Build→Build Solution** to compile the project.

5. Choose **Debug→Start Without Debugging** to execute the client.

Execute the client from the command prompt

1. Change the directory to *soa_client***/cpp/samples/WireHarness**.

2. Clean the project by typing **devenv WireHarness.vcproj /clean**.

3. Compile the project by typing **devenv WireHarness.vcproj /build**.

   This command generates **WireHarness.exe** in the debug directory.

4. Execute the client by typing **WireHarness -host "hostname"**.

   The default **-host** and **"hostname"** is **-host https://***server:port***/tc**.

Java sample SOA wire harness client

Java client

The Java sample client is located under the *soa_client***java\samples\WireHarness** directory, installed as part of your Teamcenter kit. You can build and execute this project from either the Eclipse IDE (3.2) or the command prompt with Ant(1.6.5). Before running the sample client, ensure that you have installed Wiring Harness Design Tools Integration. For a four-tier Teamcenter installation, ensure that you have Teamcenter Web tier server and Pool Manager installed. For a two-tier Teamcenter installation, ensure that you have TcServer running.

Execute the client using Eclipse IDE

1. Choose **Window→Preferences**.

   The **Preferences** dialog box opens.

2. Choose **Java→Build Path→Classpath Variables** to open the **Classpath Variable** tab.

3. Add the **TEAMCENTER_SERVICES_HOME** variable and set it to the root path of the *soa_client* folder.

4. Add the **FMS_HOME** variable and set it to the path in the **FMS_HOME** environment variable.

5. Choose **File→Import**.

The **Import** dialog box opens.

6. Choose **General→Existing projects into workspace**, and click **Next** to select an existing project.

7. Browse to *soa_client*/**java/samples/WireHarness**, select **WireHarness**, and click **Finish**.

8. Compile the project by building it.

9. Choose **Run→Debug** to open the **Debug/Run** dialog box and execute the client.

   Ensure that the project is **WireHarness** and the main class is **com.teamcenter.wireharness.Wireharness**.

Execute the client using command prompt

1. Type **>ant_buildfile build.xml** to build the client using Ant.

2. Type **>RunMe http://localhost:7001/tc** to execute the client.

Administering electromechanical objects

Administering functionality

Create a Functionality business object

If you have administrative privileges, you can create and delete functionality.

1. Launch the Business Modeler IDE application and open the **Advanced** perspective by choosing **Window→Open Perspective→Other→Advanced**.

2. Select the **Functionality** business object as the parent under which you create new child business objects.

   > Note:
   >
   > When you create a business object using **Functionality** as the parent, in addition to the new business object, you also create a master form, a functionality revision, and a revision master form.

3. In the **Business Objects** view, select the project in which you want to create the new **Functionality** business object.

4. Right-click the project and choose **Organize→Set active extension file**.

Select the **business_objects.xml** file as the file in which to save the data model changes.

5. Click **Find Business Object**, type **Functionality** in the search box, and click **OK**.

   The **Functionality** business object is selected in the **Business Objects** view.

6. In the **Business Objects** view, right-click the **Functionality** business object and choose **New Business Object**.

   The Create New Form Business Object wizard appears.

7. In the Create New Form Business Object wizard:

   a. In the **Name** box, type a name for the new business object.

   > Note:
   >
   > When you name a new data model object, add a prefix to the name to designate the object as belonging to your organization, such as a three-letter acronym.

   b. Select the **Advanced** check box if you want the new business object to derive its properties from a class. If you do this, the new business object is saved in the database as an instance of the class. Click **Browse** to the right of the **Class** box to choose the class.

      If you want to ensure that the new business object derives its properties from a corresponding class with the same name, select the **Create Primary Business Object** check box. Primary objects are business objects that have the same name as the storage class. If you choose this option, you must first ensure that the class with the same name has already been created.

   c. Click **Next**.

8. In the **Form** dialog box, define the storage class associated with the functionality master form.

9. In the **Form Storage** class pane:

   a. In the **Class Name** box, type a name for the new form storage class.

   b. In the **Parent** box, enter the class you want to be the parent of the new form storage class.

   c. Click **Add** to add an attribute to the form storage class.

      The New Attribute wizard appears.

   d. Click **Next**.

The **Business Object** dialog box appears. It displays the name of the revision to be created in the **Name** box and displays the parent of the revision in the **Parent** box. You cannot change these values.

10. Click **Next**.

11. In the **Form Storage Class** pane, define the storage class associated with the revision master form.

    a. In the **Class Name** box, type a name for the new revision master storage class.

    b. Click **Add** to add attributes to the revision master storage class.

    c. Click **Finish**.

    The new business object appears in the **Business Objects** view. A **c** on the business object icon indicates that it is a custom business object. To find the master, revision, and revision master, click **Find Business Object** at the top of the **Business Objects** view and search on the new business object name.

12. Save the changes by choosing **File→Save Data Model** or click **Save Data Model** on the toolbar. If you set the active extension file as the **business_object.xml** file, the changes are saved in that file.

13. Package your custom data model into a template for installation to a Teamcenter production server.

14. After deployment, test your new business object in the Teamcenter rich client by creating an instance of it.

    For example, in My Teamcenter, choose **File→New→Functionality**.

    Your new business object should appear in the **New Functionality** dialog box. Choose your new business object and create an instance of it.

Deleting a functionality business object

Delete existing Functionality business object

You can only delete custom **functionality** objects you have created. You cannot delete commercial-off-the-shelf (COTS) **functionality** objects, because they are standard objects provided by Teamcenter. When you delete a custom class, the associated business object is deleted. When you delete a custom primary business object, the associated class is deleted. However, if you delete a custom secondary business object, the associated storage class is not deleted. If the custom object you want to delete has children, you must delete the children before you can delete the parent. Also, when you delete an object that is referenced by other rules or objects in the system, the Business Modeler IDE tells you which references to clean up first before you can delete the selected object.

Caution:

You must use the rich client to delete all instances you created on your test server before you delete the object in Business Modeler IDE. If you delete a custom object before removing instances of that object, errors can appear in the deployment log the next time you deploy. This is because the instance is still in the database, although the object definition is removed.

1. Launch the Business Modeler IDE application.

2. Right-click the **Functionality** object and choose **Delete**.

Administering connections

Creating and deleting connections

If you have administrative privileges, you can create and delete revisable and nonrevisable connections.

Two types of revisable electromechanical connections are created during installation, **Network** and **Connection**. You can use the Business Modeler IDE application to create and delete these revisable connections.

One nonrevisable electromechanical connection **(TC_LINK)** is created during installation. You can use the Business Modeler IDE application to create and delete a business object of this nonrevisable connection.

Create a revisable business object connection

1. Launch the Business Modeler IDE application and open the **Advanced** perspective by choosing **Window→Open Perspective→Other→Advanced**.

2. Select the **Network/Connection** business object as the parent under which you create new child business objects.

   When you create a business object using **Network/Connection** as the parent, in addition to the new business object you also create a master form, a **Network/Connection** revision, and a revision master form.

3. Follow the same steps as described in Create a Functionality business object. Use the **Network/Connection** business object instead of **Functionality**.

Delete a revisable business object connection

1. Launch the Business Modeler IDE application.

2. Right-click the connection business object and choose **Delete**.

Create a nonrevisable business object connection

1. Launch the Business Modeler IDE application.

2. Select the **TC_LINK** business object as the parent under which you create new child business objects.

3. In the **Business Objects** view, select the project in which you want to create the new **TC_LINK** business object.

4. Right-click the project and choose **Organize→Set active extension file**.

   Select the **business_objects.xml** file as the file in which to save the data model changes.

5. In the **Business Objects** view, right-click the **TC_LINK** business object and choose **New Business Object**.

   The Create New Form Business Object wizard appears.

6. In the Create New Form Business Object wizard, enter the following:

   a. In the **Name** box, type the name you want to assign to the new **TC_LINK** business object.

   b. Select the **Advanced** check box only if you want to specify a different storage class from which the business object derives its properties and attributes. Choose one of the following in the **Form Storage Class** pane:

      - Click **Use new class** to create a new storage class to store the attributes.

        Type the name in the **Name** box and click **Browse** to select the new parent class.

      - Click **Use existing class** to use an existing class to store the attributes. Click **Browse** to select the new form storage class.

   c. Click **Add** to add an attribute to the form storage class.

      The New Attribute wizard appears.

   d. Click **Finish**.

      The new business object appears in the **Business Objects** view. A **c** on the business object icon indicates that it is a custom business object.

7. Package your custom data model into a template for installation to a Teamcenter production server.

8. After deployment, test your new business object in the Teamcenter rich client by creating an instance of it.

   For example, in My Teamcenter, choose **File→New→Functionality**.

   Your new business object should appear in the **New Functionality** dialog box. Choose your new business object and create an instance of it.

Delete a nonrevisable business object connection

To delete a business object of nonrevisable connection:

1. Launch the Business Modeler IDE application.

2. Right-click the **Nonrevisable Connection** business object and select **Delete**.

Administering signals

Create a signal business object

If you have administrative privileges, you can create and delete signals.

1. Launch the Business Modeler IDE application.

2. Select the **Signal/PSSignal** business object as the parent under which you create new child business objects.

   When you create a business object using **Signal/PSSignal** as the parent, in addition to the new business object you also create a master form, a **Signal/PSSignal** revision, and a revision master form.

3. Perform the same steps as described in Create a Functionality business object. Use the **Signal/PSSignal** business object instead of **Functionality**.

Delete a signal business object

1. Launch the Business Modeler IDE application.

2. Right-click the **Signal/PSSignal** business object and choose **Delete**.

Administering item elements

Create an interface business object

An *interface* is a physical port such as a connection or a terminal.

If you have administrative privileges, you can create, modify, and delete business objects of item element, including interfaces and process variables.

1. Launch the Business Modeler IDE application.

2. Select the **Interface** business object as the parent under which you create new child business objects.

3. In the **Business Objects** view, select the project in which you want to create the new **Interface** business object.

4. Right-click the project and choose **Organize→Set active extension file**.

   Select the **business_objects.xml** file as the file in which to save the data model changes.

5. Click **Find Business Object** on the **Business Objects** view toolbar and type **Interface**.

   The **Interface** business object is selected in the **Business Objects** view.

6. In the **Business Objects** view, right-click the **Interface** business object and choose **New Business Object**.

   The New Item Element wizard appears.

7. In the New Item Element wizard:

   a. Type the name you want to assign to the new business object in the **Name** box.

   b. Type a description in the **Description** box.

   c. Type the maximum number of occurrences allowed for this interface in the **Maximum Instances per Interface** box. You can enter any positive integer, 0, or –1 (infinite).

   d. Select the **Advanced** check box only if you want to choose a new class to store data for business object.

      By default, the storage class is set as the same class used by the parent business object. However, if you want to create a new class to store the data, select the **Create Primary Business Object** check box.

This creates a new class that has the same name as the new business object.

> Note:
>
> A primary business object has the same name as its associated storage class. A secondary business object uses the storage class of its parent business object. Typically, most custom business objects are secondary business objects.

e. Click **Add** to the right of the attributes table to add attributes to the business object.

f. Click **Add** to the right of the view list to choose the views to associate with this interface. To see all the available views, type an asterisk (*) in the **Find** box.

g. Click **Finish**.

The new **Interface** object appears in the tree of business objects.

8. Save the changes by choosing **File→Save Data Model** or click **Save Data Model** on the toolbar. If you set the active extension file as the **business_object.xml** file, the changes are saved in that file.

9. Package your custom data model into a template for installation to a Teamcenter production server.

10. After deployment, test your new business object in the Teamcenter rich client by creating an instance of it.

Modifying an interface business object

About modifying an interface business object

If you have administrative privileges, you can modify the definition of an existing interface, including the following information:

- The list of views in which the interface business object may be used.
- The maximum number of instances permitted in the context of a parent in a structure.
- Additional attributes that are required for this business object.

Modify an interface business object

1. Launch the Business Modeler IDE application.

2. Right-click the **Interface** business object and choose **Open**.

Details of the object appear in a new view.

3. Right-click the interface class in the view and choose **Open**.

   A new view displays the class details and the properties. Click **Add** to add attributes, or **Remove** to remove attributes.

4. To edit property rules:

   a. Right-click the selected business object in the view and choose **Open**.

      A new view displays the business object details and properties.

   b. In the properties tables, right-click the property you want to work with and choose **Edit Rule**.

      The Modify Property Rule wizard appears.

   c. Edit the rules.

   d. Choose **File→Save Data Model**.

   e. Package your custom data model into a template for installation to a Teamcenter production server.

Deleting an interface business object

Delete an interface business object

You can only delete the custom **Interface** objects you have created. You cannot delete COTS **Interface** objects because they are standard objects provided by Teamcenter. When you delete a custom class, the associated business object is deleted. When you delete a custom primary business object, the associated class is deleted. However, if you delete a custom secondary business object, the associated storage class is not deleted. If the custom object you want to delete has children, you must delete the children before you can delete the parent. Also, when you delete an object that is referenced by other rules or objects in the system, the Business Modeler IDE tells you which references to clean up first before you can delete the selected object.

> Caution:
>
> You must use the rich client to delete all the instances you have created on your test server before you delete the object in the Business Modeler IDE. If you delete a custom object before removing the instances of that object, errors can appear in the deployment log the next time you deploy. This is because the instance is still in the database, although the object definition has been removed.

1. Launch the Business Modeler IDE application.

2. Right-click the **Interface** object and choose **Delete**.

Create a process variable business object

1. Launch the Business Modeler IDE application.

2. Select the **Process Variable** business object as the parent under which you create new child business objects.

3. Perform the steps described in **Create a Functionality business object**. Use the **Process Variable** business object instead of **Functionality**.

Modify a process variable business object

If you have administrative privileges, you can modify the business object of an existing process variable.

1. Launch the Business Modeler IDE application.

2. Follow the steps described in **About modifying an interface business object** and use the **Process Variable** business object instead of **Interface**.

Delete a process variable business object

1. Launch the Business Modeler IDE application.

2. Right-click the **Process Variable** object and choose **Delete**.

Setting user exits

To allow configuration of allocations, Teamcenter provides a user exit that you can use to define the criteria by which an allocation is configured to be in or out.

Allocations may be configured independent of the source structure and the target structure that provide the context in which the allocations are created. You can release and configure allocations, using effectivity as a criterion. In cases where allocations must be configured with criteria other than effectivity, a custom object can be associated with an allocation object to capture the configuration criteria. You can define the custom object to capture the configuration condition by extending the schema. Use the **ALLOC_set_condition (tag_t allocation_tag, tag_t configuration_condition)** API to associate this custom object with an allocation.

You can then register a method with the **MECHATRONICS_is_configured_msg** user exit message and capture the business logic associated with the configuration mechanism in the registered method. An example of how to register a method with this user exit message and the implementation of the method follows:

```
METHOD_register_method("Allocation", MECHATRONICS_is_configured_msg,
                       &my_configure_allocations, NULL,&my_method))
static int my_configure_allocations (METHOD_message_t *message, va_list
args)
{
    /*Extract Allocation Tag*/
    tag_t alloc_tag = message->object_tag;
    /* Extract is_configured */
    int *is_configured = va_arg(args, int*);
    /*Temp variables */
    tag_t cond_tag = NULLTAG;
    tag_t prop_tag = NULLTAG;
    int cond_value;
    /*Initialize is_configured to true*/
    *is_configured = true;
    /*Get the condition object set on the allocation.
    This can be set using the method ALLOC_set_condition.
    Users are free to extend the schema to capture the allocation
    condition this can be a string, int or any other complex content
    including references to existing objects.*/
    ALLOC_ask_condition(alloc_tag, &cond_tag);
/*Extract properties of the condition object to determine if this
    allocation should be configured in or configured out.*/
    PROP_ask_property_by_name(cond_tag, "Configuration_value",
&prop_tag);
    if (prop_tag)
    {
        PROP_ask_value_int(prop_tag, &cond_value);
        if (cond_value == 0)
            *is_configured = false;
        else
            *is_configured = true;
    }
    return ITK_ok;
}
```

Using a Multi-Site environment

You can use the Wiring Harness Design Tools Integration in a *Multi-Site Collaboration* environment to share wire harness designs with other sites. You can:

- Export wire harness designs that you own from your site to a remote site.
- Import wire harness designs from a remote site.
- Transfer ownership of wire harness designs you own to a remote site.
- Check out and check in wire harness designs from a remote site.

- Publish wire harness designs.

In most cases, there are no additional configuration or administration requirements for integration. However, if you want to publish wire harness or allocation information, you can set the **TC_publishable_classes** preference to **Item**, **Dataset** or **Form**, as appropriate. By default, only items are published, unless you set this optional preference.

Identifying user name and password for Capital tool using custom exit

If your site uses a security system other than Teamcenter Security Services, for integration between Teamcenter and *Capital*, you can use the **USER_get_mentor_capital_credential** custom exit to identify the user name and password for *Capital*.

Users can define the logic for the custom exit. The custom exit is used to provide the user credentials for the *Capital* system to connect with Teamcenter for workflow Integration using the *Capital* web services.

The exit is used by the following workflow handlers for workflow integration between Teamcenter and *Capital*:

- HRN-capital-set-release-state

- HRN-capital-set-reject-state

5. Wiring Harness APIs

Wire Harness APIs

The Teamcenter Integration Toolkit (ITK) API support is extended to support wire harness design objects. These APIs are organized per module and allow you to create, edit, and delete various wire harness design objects including connections, ports and terminals, signals, route objects, allocations, and electrical product structures.

PSCONN module

The **PSCONN** module provides APIs that allow the manipulation of connectivity data. You can use these APIs from an external system to create, edit, and delete connections. The APIs are defined in the **psconnection.h** header file.

The **Connection** object can be managed, revised, released, and configured. This object is a subclass of **Item** and therefore supports all the functionality supported by **Item**. You can create a subtype for a connection and customize the behavior of the subtype to suit specific user needs.

Teamcenter provides two predefined subtypes for use when modeling wire harness:

- **Connection**

  Use this connection type to model logical or physical connections.

- **Network**

  Use this connection type to model functional networks.

Each **Connection** object has a unique ID similar to **Item** objects. After it is defined, it can be instantiated multiple times in an electrical product structure. Currently, there are no restrictions on the type of view in which a particular subtype of a connection can be instantiated. You can combine an electrical view with a mechanical view and instantiate all electrical and mechanical components in a single view. However, Siemens Digital Industries Software recommends that you separate the electrical product structure from the mechanical product structure to obtain maximum benefit from the Teamcenter data model.

You can associate instances of a connection object to other instances or context-specific occurrences of electrical components (such as wires, cables, and signals) in the product structure, using the following relationships:

- **TC_Connected_To**

  Defines the connectivity of a connection object by associating the occurrences of connecting ports or terminals.

- **TC_Implemented_By**

  Associates an electrical device with a connection that it implements. For example, you can associate a specific occurrence of a wire or cable with an instance of a connection in an electrical structure by means of the **implementedBy** relationship.

- **SIG_asystem**

  Associates an occurrence of a connection with a specific instance of a signal in an electrical structure.

The Teamcenter data model also allows you to model non-revisable lightweight connections as **GDELink** connections. In the context of wire harness modeling, **GDELink** is best used to model connectivity that is internal to an electrical connector or device. The API associated with **GDELink** is defined as part of the **GDE** module, described next in this chapter.

You can find code examples for creating a connection, instantiating it in an electrical view of a product, and defining the connectivity of the connection to terminals in the sample program. The sample program is located in the Mechatronics folder of the Teamcenter sample files directory.

GDE module

The **GDE** module provides APIs that allow the manipulation of item elements. You can use these APIs from an external system to create, edit, and delete GDEs or instances of GDEs. The APIs are defined in the **gde.h** header file.

The **GDE** object can be managed and released, but not revised. This object is a subclass of **WorkspaceObject** and therefore supports all the functionality supported by **WorkspaceObject**. You can create a subtype for a GDE and customize the behavior of the subtype to suit specific user needs.

Teamcenter provides three predefined GDE subtypes.

- **Connection_Terminal**

  Use this subtype of **GDE** to model physical, electrical terminals of an electrical connector or device.

- **Network_Port**

  Use this subtype of **GDE** to model ports of a function in an electrical functional model.

- **ProcessVariable**

  Use a process variable as a parameter to control or monitor a process. In the context of wire harness modeling, process variables define the message types carried by signals.

GDEs also support decomposition and you can use it to model an interface consisting of a set of ports or terminals.

You can place restrictions on the predefined GDE subtypes that are valid for defining the connectivity of a specific connection type in the product structure. Also, you can associate GDE types to view types to restrict the usage of those types to specific views. You can place further restrictions on the number of instances of each GDE object that may be created in an item.

You can build relationships between instances of GDEs to model scenarios in which the interface provided by an occurrence of a GDE is realized by another occurrence of a GDE at a higher level in the electrical assembly. You can model this relationship with a **TC_Realized_By** relation in the context of an electrical structure.

SIGNAL module

The **SIGNAL** module provides APIs that allow the manipulation of signals. You can use these APIs from an external system to create, edit, and delete signals. The APIs are defined in the **pssignal.h** header file.

The **Signal** object can be managed, revised, released, and configured. This object is a subclass of **Item** and therefore supports all the functionality supported by **Item**. You can create a subtype for a signal and customize the behavior of the subtype to suit specific user needs.

A signal carries a message between functions or electrical components. You can model the nature of the message carried by a signal by associating a process variable to the signal in the context of an electrical structure. To add a signal to an electrical structure, instantiate a signal and then define the various properties and relationships associated with the signal, including signal source, signal target, and associated systems.

You can relate one signal with another signal by decomposition, substitution, and redundancy relationships.

- **Decomposition**

  This relationship type allows users to build a signal object as an assembly of signal components. The association between the signal objects is of the parent-child type, where the parent signal is called the *relating* signal and the child signal is called a *related* signal. This relationship type is similar to the **composition** relationship between item objects.

- **Substitution**

  This relationship type allows users to define a substitute signal to replace the related signal in a given product structure. This allows a user to create alternate configurations of a system when several choices of signal objects are available. This relationship type is similar to the **alternate** relationship between item instances.

- **Redundancy**

  This relationship type allows users to define a relationship where the related signal is replicated by the relating signal. The **redundant** relationship is the association between the two instances of the signal; it indicates that the related object is a *redundant alternative* to the primary signal in a given context.

You can also associate a signal with another physical entity that transmits or processes the signal. The physical entity for processing and transmitting the signal is called the *associated system*, and the relationship between the objects is called *association*. Objects associated to signals with associated system relationships also have a role associated with them, such as source, target, and transmitter.

ROUTE module

The **ROUTE** module provides APIs that allow the manipulation of routes. You can use the APIs from an external system to create, edit, and delete routes. The APIs are defined in the **route.h** header file.

The **ROUTE** module provides APIs to work with the following types of objects.

- **RoutePath**

  A **RoutePath** object represents the physical path associated with a wire or cable in an electrical product. You can define an instance of a **RoutePath** object as a sequence of **RouteSegment** instances or a sequence of **RouteNode** instances.

  > Note:
  >
  > When you define a **RoutePath** instance as a list of **RouteNode** instances, you can associate a **RouteCurve** instance with the **RoutePath** instance to represent the 3D curve passing through the **RouteNode** instances.

- **RouteSegment**

  A **RouteSegment** object defines a segment of the path associated with a wire or cable. You define the segment with a start node, an end node, and a **RouteCurve** instance.

- **RouteNode**

  A **RouteNode** object is a reference point that is used to define a **RouteSegment** object or a **RoutePath** object.

- **RouteCurve**

  The curve information associated with a **RouteSegment** or **RoutePath** object is captured as a **BSplineCurve** object. Use the **RouteCurve** object to capture the geometrical definition of a **BSplineCurve** object.

You can use these APIs to model the physical route associated with a wire or cable in an electrical structure. The Teamcenter route model is a simplified implementation of the AP212 route model and there are no separate objects to define topological and geometric data associated with routes. Also, the Teamcenter model defines all route objects only in the context of a given electrical structure and these objects cannot be shared across multiple structures.

In the context of an electrical structure, you can associate **RoutePath** instances with instances of wire or cable using the **TC_Routed_By** relationship. This captures the 3D physical path associated with the wire instance in the electrical structure.

ALLOC module

The **ALLOC** module provides APIs that allow the manipulation of allocations. You can use these APIs from an external system to create, edit, and delete allocations. The APIs are defined in the **allocation.h** header file.

The **ALLOC** module provides APIs that allow you to manage allocations between components of two different view types of a product. For example, you can use an allocation object to model the association between a function in the functional view and a physical connector or device in the electrical view of a product. The Teamcenter model supports allocations between components of any two views of a product. It is not required that the allocated views belong to the same item. You can allocate a component in one view to one or more components in a different view.

You can create allocation subtypes and customize the behavior of the subtypes to meet user-specific requirements. You can group allocations defined between two views with an **AllocationMap** object. The **AllocationMap** object allows you to establish the source and target view contexts for defining the allocations. The **AllocationMap** object is subclassed from **Item** and can be revised.

In the context of wire harness modeling, you can use allocations to model the associations between electrical components; these associations may be from functional to logical and logical to physical product views. For example, you can associate functions from the functional model with physical connectors or devices in the physical electrical model. Similarly, you can allocate networks to physical connections and ports to physical terminals.

6. Integrating with electromechanical applications

Integrating with electromechanical applications

The integration of electromechanical authoring applications with Teamcenter allows you to represent an electrical product structure (including the connectivity data) using the Teamcenter data model. After you create the electrical product structure in Teamcenter, you can use this data with other Teamcenter features such as process management and configuration management.

Teamcenter also allows integration with various electrical 3D CAD packages capable of working with this connectivity data; for example, NX Routing. The integration allows users to create unified electrical product structures, including both electrical and mechanical aspects of electromechanical products, and the allocations between them.

Harness design process

Harness design process flow

The following steps define the typical sequence of operation while working with integration between Teamcenter, ECAD, and MCAD systems:

1. Electrical designer authors electrical connectivity information in the ECAD application.

2. Electrical designer publishes the electrical design data to Teamcenter.

    a. Prepares the data.

    b. Transfers the data to Teamcenter.

3. When the data is published to Teamcenter, you can create a workflow and apply an appropriate release status to this data.

4. The mechanical engineer accesses data from Teamcenter to add more information.

    a. Prepares the data.

    b. Transfers information from Teamcenter to MCAD.

    c. Uses a 3D electrical application (such as NX Routing) to add more details to the electrical data; for example, wire lengths and route data.

    d. Publishes the updated data to Teamcenter.

5. Electrical designer uses the ECAD application to search for published data in Teamcenter and retrieve the data modified by the mechanical designer.

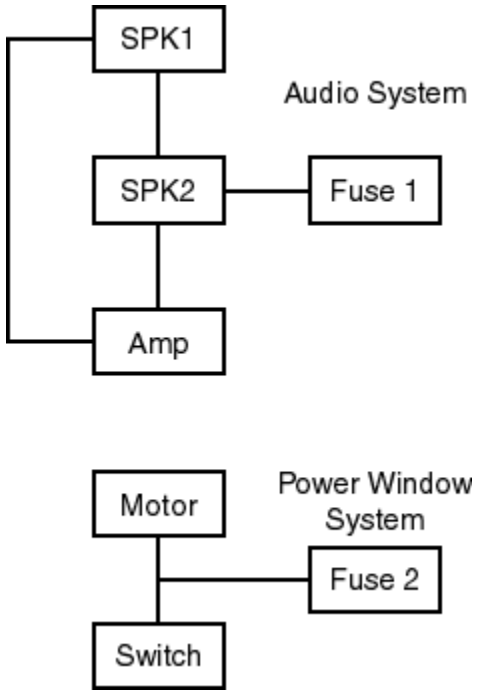
6. Electrical designer uses the ECAD application to reconcile the changes made to the connectivity data and make additional edits if necessary.

7. Electrical designer publishes the finalized data and schematic datasets into Teamcenter.

Authoring electrical connectivity information in ECAD

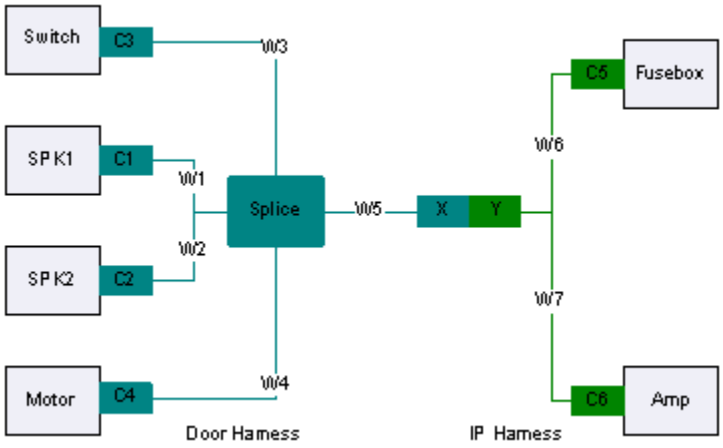
The harness designer typically creates functional design, physical design, or harness design in the ECAD system. The functional model is an abstract model of the system to be designed containing all the system components and the logical connections between them.

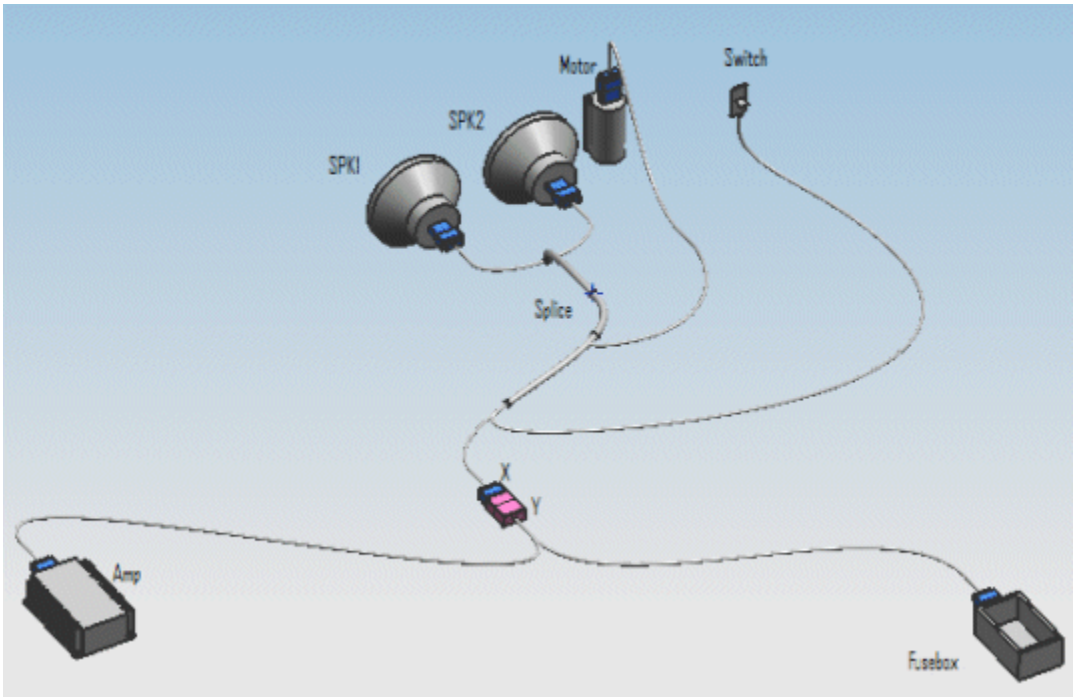
For example, the following figure shows the design view of an audio system.



The harness design contains the electrical components corresponding to physical components and the electrical connectivity information between them. The harness design of an audio system is shown in the following figure.

The physical design is shown in the following figure.





For more details about functional and harness design, see your ECAD user manual.

Tip:

Because the design view, harness view, and other views associated with an electromechanical product represent various aspects of the same product, you must identify a top-level product that ties all these views together. You then publish this top-level product to Teamcenter with the associated views. Typically, you define the top-level product early in the design process in ECAD applications when you create a project to design the connectivity data. The top-level product must be modeled in Teamcenter as an item; you can choose to represent the top-level product as a

specific subtype of an item. You can specify a unique identifier for the item in the ECAD application or Teamcenter can automatically generate a unique identifier when the item is published to it.

Publishing electrical design data to Teamcenter

Publishing design data

After the design is completed in the ECAD system, you can publish this information to Teamcenter and share with other applications (such as MCAD) or work with Teamcenter features such as Workflow.

You can publish this information using published wire harness SOA services. External ECAD and MCAD applications can create, modify wire harness structure with variants and options and get both the configured and unconfigured wire harness structure information.

You can also use the PLM XML (manual or SOA) to publish design data.

Prepare the data

Write a PLM XML file for the electrical design data created in the ECAD system. For more information about writing a PLM XML file, see the PLM XML SDK documentation at:

http://www.plm.automation.siemens.com/en_us/products/open/plmxml/index.shtml

Note:

- Make sure the PLM XML SDK version is compatible with the Teamcenter version to which the data is exported.
- Make sure that the electrical design data is mapped to the correct PLM XML element.
- Make sure to use the correct transfer mode (the one designed for exporting ECAD data to Teamcenter) when generating the PLM XML file.

Transfer the data to Teamcenter

You must transfer the generated PLM XML file to Teamcenter to create the appropriate Teamcenter objects/structure.

The PLM XML file can be transferred to Teamcenter in two modes:

- Manually by using Teamcenter PLM XML import mechanism.
- Automatically by the integration between the ECAD system and Teamcenter, using the Application Integration (AI) service framework.

The AI service is part of the service-oriented architecture (SOA) used by Teamcenter. The AI service allows Teamcenter and an external application to share data. You can use this framework to integrate any ECAD application with Teamcenter. The data can be transferred to and from Teamcenter in PLM XML format.

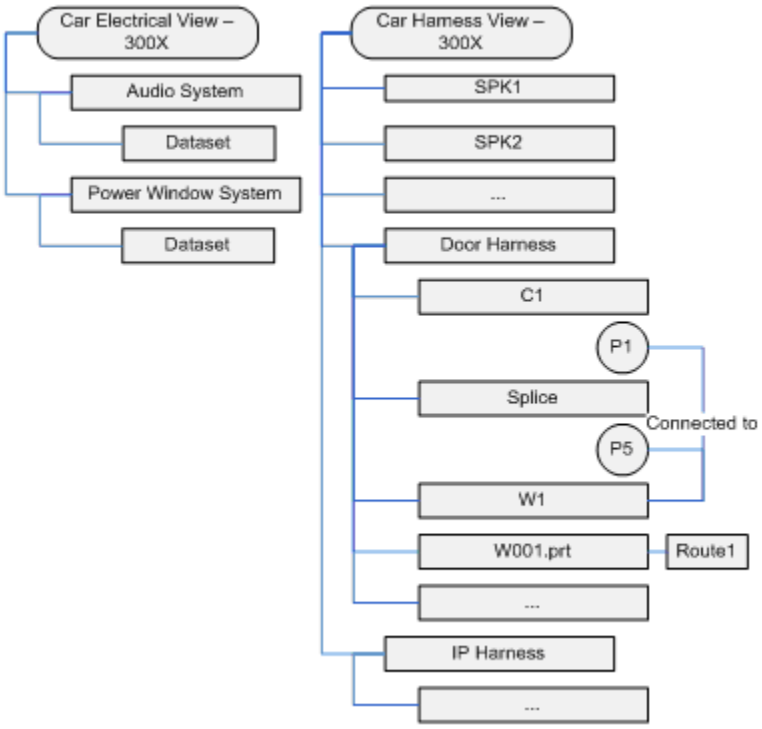
To communicate with the AI service, you must create a client that resides on the external ECAD application and communicates with the common Web service. The client may be written in C++ or Java.

The client does the following:

- Provides services used by the Teamcenter Web server and applications.
- Sends published requests and their associated PLM XML data and dataset files or streams to the AI service.
- Takes synchronization requests and their associated PLM XML data and dataset files or streams from the AI service.
- Uses SOAP as the access protocol.
- Allows users of the Resource Browser application to browse resources in the external application.

Working with the data in Teamcenter

After the data is published to Teamcenter, the information is available as Teamcenter components. For example, the audio system you designed will have two structures representing the design view and harness view in Teamcenter as shown in the following figure.



After the electrical data is published in Teamcenter, it can be submitted to a workflow process for approval and configuration management. The data published to Teamcenter can also contain various options and variants created in the ECAD application.

Sharing ECAD data with other applications

ECAD data in Teamcenter can be accessed by other applications such as NX Routing for updating electrical data— for example, wire length or route data—and to create a physical assembly with components that implements the electrical functionality.

The data exchange between Teamcenter and other ECAD and MCAD applications follows the same process as described in *Publishing design data*. The integration between Teamcenter and NX allows data exchange without this process.

After you publish the MCAD data to Teamcenter, you have three different views—design, harness, and physical.

- *Design* views represent various subsystems of the product, for example, a music system or braking system. These views may contain datasets with attached graphical diagrams in SVG format. The diagrams may contain functional connectivity diagrams or physical wiring connection diagrams.

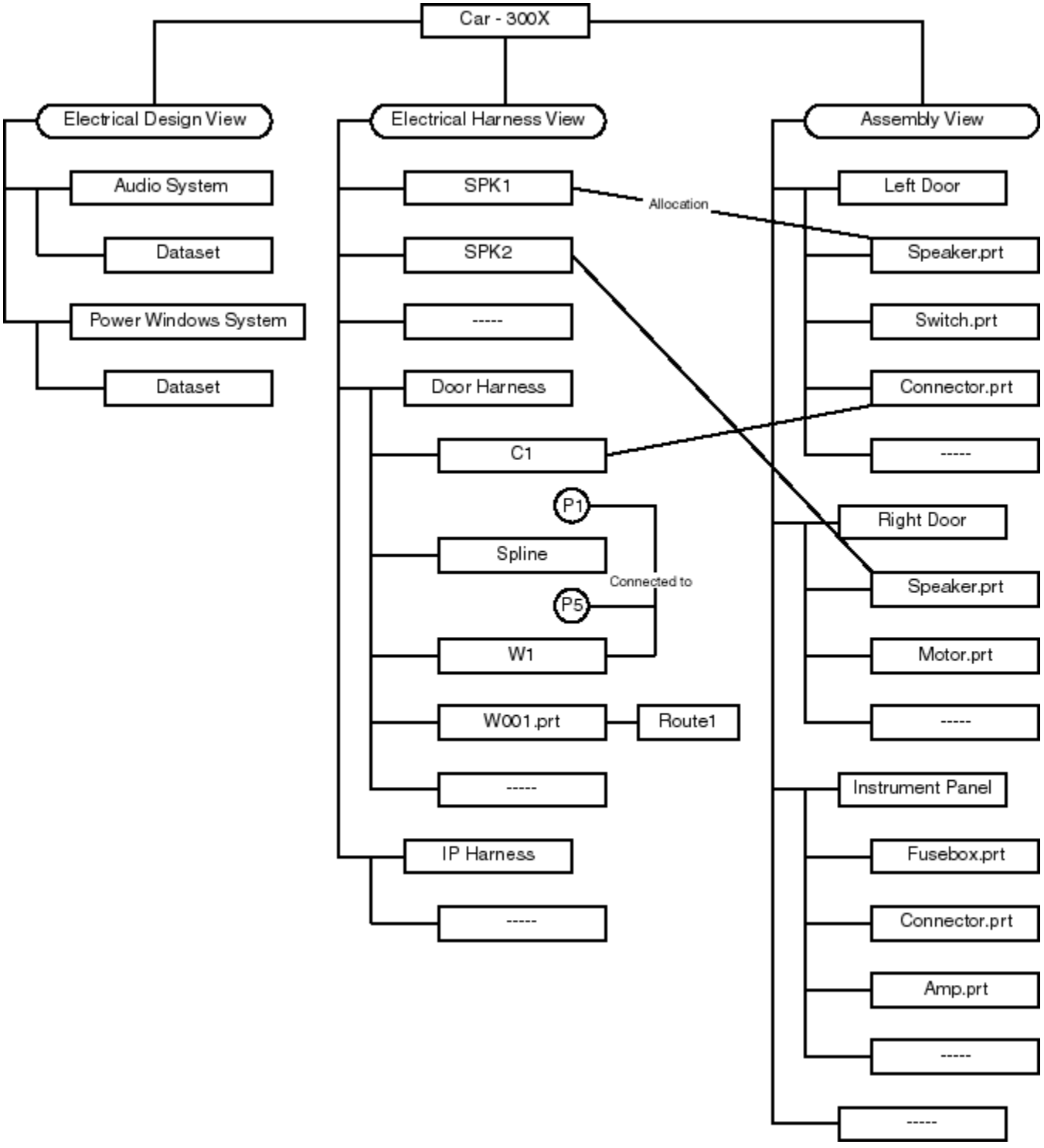
  Design archive dataset types help manage electrical design archive files. Design archive dataset supports zipped versions of electrical design archive files as named references.

- *Harness* views model one or more harnesses and electrical connectors or devices. These views represent the tangible implementation of the functional design at an appropriate level of granularity for the product; for example, a left door harness or an instrument panel harness. You initially populate this view from the ECAD application with connectivity information and logical device connectors. A 3D MCAD application such as NX Routing subsequently adds physical data and completes the assignment of logical connectors to physical parts.

- *Physical* views contain 3D representations of physical devices and connectors. These views represent the packaging information associated with the product. The top-level item of this view should contain all the devices that the harness must be routed to or around. This view is typically populated from a 3D MCAD or electrical application such as NX Routing.

Typically, each project in an ECAD application includes one or more designs that comprise functional or logical electrical physical electrical models. It also contains an electrical bill of materials. For example, a vehicle project may include a design for each individual harness that is fitted in the vehicle.

A typical representation of a unified electromechanical product structure is shown in the following figure. This representation also indicates the allocation relationships between the electrical harness view and the mechanical assembly view.



Reconciling the Teamcenter data with additional electrical information

After the data is updated in Teamcenter, the ECAD and MCAD system can access this published information for adding additional electrical information. This process can iterate until the desired goal is met. When the design intent is met, the product structure can be released for manufacturing.